

POLITECNICO DI TORINO

LM-32 (DM270) - Computer Engineering (cybersecurity)
Department of Control and Computer Engineering (DAUIN)

Master of Science's Thesis

Full-Lifecycle API Management: Microgateway Infrastructural Pattern adopting Kong Gateway



Supervisor

Prof. Riccardo Sisto

Co-Supervisor

Prof. Fulvio Valenza

Candidate

Davide Arcolini

Liquid Reply Tutors

Dr. Pietro Santoro

Dr. Gianmario Colli

Dr. Andrea Moscato

Academic Year 2022-2023

*To my mother, my father and my sister, Giulia,
for your endless love and support.*

Summary

In today's digital age, **Web APIs** (Web Application Programming Interfaces) have emerged as a fundamental technology, enabling secure interactions between organizations and their customers or partners. These APIs facilitate data sharing, third-party extensions, and mobile application integration, but their accessibility and connectivity underscore the critical importance of security and data integrity. To address security concerns, organizations have adopted API Gateways, pivotal in securing Web APIs interactions. Acting as intermediaries in an organization's digital infrastructure, these gateways manage incoming requests, data manipulation, routing, and traffic control, with a strong focus on backend service security. Traditionally centralized, API Gateways were **designed with a monolithic approach and managed by dedicated teams**, but as the landscape of Web APIs development evolves, a new approach, the **Microgateway Infrastructural Pattern**, is introduced. API Microgateways provide a lightweight, flexible, and decentralized solution for microservices integration, empowering developers to focus on enhancing their individual microservices without the complexities of centralized API gateway management.

The primary objective of this thesis revolves around the definition of an **API Microgateway Pattern**. Specifically, the aim is to study, develop and showcase realistic **API microgateway scenarios** employing **Kong Gateway** open-source (OSS) and Kong Gateway Enterprise. In alignment with industry recognition, as stated by [Gartner](#), Kong stands as the preeminent leader in the API Gateway landscape. The focus of this research is to comprehend and analyse diverse requirements for adopting Kong Gateway as a microgateway technology. While these deployments are orchestrated within Kubernetes clusters, it is intended to show how, in a microgateways scenario, businesses can integrate authentication with external Identity Providers like Keycloak, security and authorised access for the APIs consumers, logging, traffic monitoring, rate limiting and other substantial aspects of the APIs management. Finally, the objective extends to the integration of these proof of concepts into a holistic DevSecOps scenario, thus embracing best practices for **Full-Lifecycle API Management**.

The following list presents the objectives of this thesis.

1. **Definition of the API Microgateways Pattern and Kong integration.** The first objective revolves around the concept of the Microgateway Infrastructural Pattern. This study seeks to elucidate the rationale behind this pattern and explore its seamless integration with the Kong product. In the process, a deep dive into the workings of Kong Gateway is carried out, in order to develop cutting-edge methodologies to manage Kong Gateway as a microgateway technology.
2. **Realistic PoC scenarios with Kong.** The second objective entails the practical implementation of realistic and enterprise-ready Proof of Concept (PoC) scenarios using Kong microgateways. These scenarios should encompass essential aspects such as authentication, authorization, monitoring, and more, all orchestrated within Kubernetes

deployments.

3. **DevSecOps integration with Kong microgateway.** The final objective focuses on automation and integration of the Microgateway Infrastructural Pattern with DevSecOps best practices, adopting Kong Gateway as the microgateway technology. Automation servers like Jenkins and GitHub Actions are employed to illustrate the seamless merging of these concepts, emphasizing the importance of adhering to best practices in Full-Lifecycle API Management.

This research provides a comprehensive description of the Microgateway Infrastructural Pattern within the integration of the commercial product Kong Gateway. The second, and in particular, the third chapter, presents the analyzed background and defines clearly the paradigm developed in accordance with Kong. This accomplishes the **first objective**.

To accomplish the **second objective**, the initial step involved designing and developing a robust yet comprehensive gradle-based Spring Boot CRUD application, written in Kotlin. This application served as the foundational element, allowing complete control over the APIs, thus facilitating integration with specific gradle-based tools that played a pivotal role throughout the project. These tools encompassed essential components such as JUnit for testing, OpenAPI Specifications for precise API documentation, Google Jib for streamlined containerisation, and more. The Spring Boot application designed as a lightweight microservice following best practices, intentionally omits certain functionalities such as authentication, authorization, and comprehensive monitoring. This approach aligns with the microservice pattern, where the role of this specific service focuses solely on the management of resources. To address this, the deployment of the microservice alongside instances of Kong microgateway, both the open-source version and Kong Gateway Enterprise, have been evaluated. This exploration involved an array of plugins tailored to suit each unique use-cases considered, spanning various authentication scenarios (OAuth 2.0, OpenID Connect, SAML 2.0) integrating external Identity Providers and Authorization Servers like Keycloak, as well as policy enforcement tools like Open Policy Agent (OPA). Additionally, Prometheus and Grafana are employed to monitor the status of the cluster, while HashiCorp Vault is utilized for secret management.

Finally, the **third objective** has been accomplished by the integration of DevSecOps principles with Kong, resulting in a full lifecycle API management for the showcased scenarios. Powerful automation tools such as Jenkins and GitHub Actions were leveraged, allowing fast and secure CI/CD pipelines to be established. The entire software development lifecycle, from source code management to comprehensive testing, was efficiently managed by these pipelines. Tagged Docker images were automatically built, tested, and pushed to a designated Docker Registry, making them ready for deployment to the Kubernetes cluster. Subsequently, the generation and validation of OpenAPI Specifications and Security Audits against 42Crunch servers were orchestrated by the pipelines. Configuration of Kong microgateway instances was also automated using the deck automation tool, ensuring seamless and consistent deployment across different environments. Upon completion of the automated processes, the solutions were deployed to a dedicated test environment within the Kubernetes cluster. Here, functional tests were conducted using tools such as Insomnia and Postman to ensure the correctness of the APIs, while stress tests were executed using Gatling to assess performance under heavy loads. The entire deployment process was closely monitored by Prometheus servers, continuously scraping data from Kong instances, thereby providing valuable insights presented through Grafana dashboards, ensuring the reliability and efficiency of the integrated solutions.

All the PoC scenarios developed in this research has undergone a series of functional and performance tests using Postman, Insomnia and Gatling. Kong microgateways have been

deployed and tested both in a on-premise microk8s cluster and a on-cloud Azure Kubernetes Service (AKS) cluster, where OAuth 2.0, OpenID Connect and SAML 2.0 authentication and authorization flows have been throughout validated.

Acknowledgements

I began this research in April 2023, and since then, many people have supported and guided me through the months that followed. First and foremost, I want to express my gratitude to my two academic supervisors, Professor Riccardo Sisto and Professor Fulvio Valenza, for giving me the opportunity to complete this thesis and for sharing valuable insights. I also want to acknowledge my corporate tutors and everyone at Liquid Reply who welcomed me and never missed a chance to help me when I was in need. In particular, a special thanks goes to Gianmario Colli, whose professionalism and kindness were crucial in motivating and inspiring me. Thanks also to Andrea Moscato and Pietro Santoro for following me on this journey. Also, I would like to extend my appreciation to Lucas Buschlinger from Spike Reply, who provided the tools and capabilities that helped achieve the results presented in this thesis.

Moreover, this thesis would not have been possible without the constant support of my closest friends. I want to express my gratitude to each of you for inspiring me and bringing joy during challenging times. Thanks to my life-long friends, Michele and Pietro, with whom I want to share my joy in concluding this chapter of my life that I have always waited for. Thanks to Filippo, whose wisdom always provided valuable insights during this time. Thanks to my friends, Lorenzo, Luca, and Matteo, with whom I shared days and nights in these challenging months. Thanks to the new life-long friends I met during these years and the friends from HKN, Edoardo, Claudio, Manuel and Francesca. We built amazing relationships, and my joy is shared with you as well. Thanks also to my roommates, Alexander and Davide, for sharing my lows and highs during this period. A big thanks to Elena, whose great potential and positive energy have been sources of inspiration for me and will always be as such. Thanks to my old friend Chiara, as we both shared these weeks in writing our master's theses, supporting each other every day. Finally, thanks to Marta, for making this time less taxing with her stories and laughs.

Last but not least, I want to express my gratitude to my whole family. None of this would have been possible without your support, as you always rooted for my choices and trusted me in pursuing my objectives. You have always been my number one supporters, and I will always be indebted for such. Thanks, mamma, papà, and Giulia. I love you all.

Contents

1	Introduction	1
1.1	Objectives of the thesis	1
1.2	Structure of the thesis	2
2	Full-Lifecycle API Management	4
2.1	Application Program Interface (API)	5
2.1.1	A bit of history	5
2.2	Understanding Web APIs	6
2.2.1	The impact of Web APIs to the modern internet	6
2.2.2	How to interact with Web APIs	7
2.2.3	Protocols and Standards	8
2.2.3.1	Types of Web APIs by audience	8
2.2.3.2	Types of Web APIs by architecture	9
2.2.3.3	Protocols for Web APIs	10
2.3	The API Lifecycle	10
2.3.1	The Producer Lifecycle	11
2.3.1.1	Blueprinting APIs	11
2.3.1.2	Implementing APIs	12
2.3.1.3	Exposing APIs	12
2.3.2	Swagger OpenAPI Specification	13
2.3.2.1	Structure of the OAS file	13
2.3.2.2	Benefits of adopting OAS	14
2.4	Development, Security and Operations (DevSecOps)	15
2.4.1	End-To-End Automation with APIOps	16
2.4.1.1	Employing CI/CD pipelines	17
2.4.1.2	Shifting to Infrastructure as Code (IaC)	18
2.4.2	API Security by Design with 42Crunch	18
3	Microgateway Infrastructural Pattern	20
3.1	The rise of microservices	21
3.1.1	A bit of history	22
3.1.2	From monolith to microservices	23
3.1.2.1	The strangler pattern	25
3.1.2.2	Hexagonal architecture	25
3.1.2.3	Benefits and drawbacks	27
3.1.3	Orchestrate microservices with Kubernetes	27
3.1.3.1	Introduction to K8s	28
3.1.3.2	Managing microservices concerns	29

3.2	Exposing APIs on the internet	30
3.2.1	The traditional approaches	30
3.2.1.1	Direct client-to-microservices communication	31
3.2.1.2	API Gateway Pattern	31
3.2.2	A new approach adopting microgateways	33
3.2.2.1	API Microgateways Pattern	33
3.2.3	Comparison and considerations	35
3.3	Adopting Kong as the microgateway technology	36
3.3.1	Kong Gateway	38
3.3.1.1	How it works	38
3.3.1.2	Admin APIs	40
3.3.1.3	Declarative configurations	40
3.3.2	Kong Plugin Hub	42
3.3.3	Microgateway approach in DevSecOps	44
3.3.3.1	decK automation tool	45
3.3.3.2	Integration with Kubernetes	47
4	Enterprise Use-Cases: Exposing APIs with Kong microgateway	49
4.1	MoviesCatalog: a simple CRUD microservice	50
4.1.1	The application	50
4.1.1.1	Core plugins	50
4.1.1.2	APIs	51
4.1.2	Tasks automation with Gradle	52
4.1.2.1	JUnit 5 tests	52
4.1.2.2	Google Jib	54
4.1.2.3	OpenAPI Specification file	55
4.1.2.4	Performance tests with Gatling	56
4.2	Scenario: manage confidential information	58
4.2.1	Cluster infrastructure	58
4.2.1.1	Configuring HashiCorp Vault	59
4.2.1.2	Storing client secrets in HashiCorp Vault	60
4.2.2	Kong microgateway	61
4.2.2.1	Configuring Vault entity	61
4.3	Scenario: integrate observability	62
4.3.1	Cluster infrastructure	62
4.3.1.1	Configuring Prometheus	63
4.3.1.2	Configuring Grafana	65
4.3.2	Kong microgateway	66
4.3.2.1	Configuring Prometheus plugin	67
4.4	Scenario: implement Identity and Access Management (IAM)	67
4.4.1	Cluster infrastructure	69
4.4.1.1	Configuring Keycloak	70
4.4.1.2	Configuring Open Policy Agent	72
4.4.2	Kong microgateway	74
4.4.2.1	Configuring JWT plugin	75
4.4.2.2	Configuring OPA plugin	76
4.4.2.3	Configuring OIDC plugin	76
4.4.2.4	Configuring SAML plugin	78
4.4.3	Running the scenarios	79
4.4.3.1	OAuth 2.0 - Authorization Code Flow	80

4.4.3.2	OAuth 2.0 - Client Credentials Flow	82
4.4.3.3	OAuth 2.0 - Resource Owner Password Flow	84
4.4.3.4	OpenID Connect - Authorization Code Flow	86
4.4.3.5	SAML 2.0 - Service Provider (SP) Initiated Login	88
5	Enterprise Use-Cases: APIOps with Kong microgateway	91
5.1	Automation tools and infrastructure	92
5.1.1	Infrastructure technologies	93
5.1.2	Automation tools	94
5.2	Continuous Integration (CI)	95
5.2.1	Unit and integration phase	97
5.2.1.1	JUnit 5	97
5.2.1.2	Google Jib	97
5.2.2	APIs management phase	98
5.2.2.1	OpenAPI Specification file	98
5.2.2.2	Security Audits with 42Crunch	99
5.2.3	Kong microgateway management phase	101
5.2.3.1	Converting OAS to Kong and validating	102
5.3	Continuous Delivery (CD)	102
5.3.1	Deployment phase	103
5.3.1.1	Rolling-Out Kong microgateways	103
5.3.2	Test phase	105
5.3.2.1	Functional tests with Insomnia	105
5.3.2.2	Functional tests with Postman	105
5.3.2.3	Performance tests with Gatling	106
6	Enterprise Use-Cases: Test Result and Architectures Validation	108
6.1	Introduction	109
6.1.1	Objectives of the thesis	109
6.1.2	Work recap	110
6.2	Test and validation methodologies	111
6.2.1	Continuous Integration and Continuous Delivery (CI/CD)	112
6.2.2	Functional Tests	113
6.2.3	Performance Tests	113
6.2.4	Real-Time Monitoring	114
6.3	Presentation of the results	114
6.3.1	Description of the scenario	114
6.3.2	Results and considerations	117
6.3.2.1	Continuous Integration and Continuous Delivery (CI/CD)	117
6.3.2.2	Functional Tests	120
6.3.2.3	Performance Tests	121
6.3.2.4	Real-Time Monitoring	121
6.3.3	Analysis of the results	123
7	Conclusions and Future Work	125
	List of Figures	126
	Bibliography	130
	Sitography	132

1. Introduction

Securing the exposure of services over the internet has always been a paramount concern for businesses and organizations. In this digital age, **Web APIs** (Web Application Programming Interfaces) have emerged as a fundamental technology, facilitating secure interactions between organizations and their customers or partners. Web APIs enable software applications to communicate across the internet, playing a pivotal role in data sharing with partners, empowering third-party developers to create extensions, and enabling mobile applications to interact with backend systems. However, the ease of access and connectivity also demands an increased focus on **security and data integrity**.

As businesses and organizations open their services via Web APIs, they establish potential access points that may be targeted by cyberattacks. This inherent exposure necessitated the development and adoption of a significant technology, known as the **API Gateway**. These gateways play a pivotal role in supporting Web API security, acting as intermediaries positioned at the forefront of an organization's digital infrastructure. They perform diverse functions, encompassing the handling of incoming requests, data transformation and manipulation, routing requests to their intended destinations, and comprehensive traffic flow management. Furthermore, their central role lies in ensuring the robust security of backend services. Traditionally, API Gateways have been centralized components within an organization's IT framework, strategically positioned and often managed by dedicated teams. However, in response to the evolving landscape of Web API development and the prevalence of microservices, a new infrastructural paradigm, the **Microgateway Infrastructural Pattern**, based on API Microgateways, has been studied and it will be presented in this research.

API Microgateways represent a lightweight, flexible, and dedicated approach that can be easily integrated in microservices infrastructures. They possess the capability to execute the same functions as a centralized gateway. Nevertheless, under this pattern, the management of Web APIs becomes entirely decoupled among teams. This empowers developers to concentrate on the production and refinement of their individual microservices without being entangled in the complexities of centralized API gateway management.

1.1 Objectives of the thesis

The primary objective of this thesis revolves around the definition of an **API Microgateway Pattern**. Specifically, the aim is to study, develop and showcase realistic **API microgateway scenarios** employing **Kong Gateway** open-source (OSS) and Kong Gateway Enterprise. In alignment with industry recognition, as stated by Gartner [Sha22], Kong stands as the preeminent leader in the API Gateway landscape. The focus of this research is to comprehend and analyse diverse requirements for adopting Kong Gateway as a microgateway technology. While

these deployments are orchestrated within Kubernetes clusters, it is intended to show how, in a microgateways scenario, businesses can integrate authentication with external Identity Providers like Keycloak, security and authorised access for the APIs consumers, logging, traffic monitoring, rate limiting and other substantial aspects of the APIs management. Finally, the objective extends to the integration of these proof of concepts into a holistic DevSecOps scenario, thus embracing best practices for **Full-Lifecycle API Management**.

The following list presents the objectives of this thesis.

1. **Definition of the API Microgateway Pattern and the integration with Kong.** The first objective revolves around the concept of the Microgateway Infrastructural Pattern. This study seeks to elucidate the rationale behind this pattern and explore its seamless integration with the business product Kong. In the process, a deep dive into the workings of Kong Gateway is carried out, in order to develop cutting-edge methodologies to manage Kong Gateway as a microgateway technology.
2. **Realistic PoC scenarios with Kong.** The second objective entails the practical implementation of realistic and enterprise-ready Proof of Concept (PoC) scenarios using Kong microgateways. These scenarios should encompass essential aspects such as authentication, authorization, monitoring, and more, all orchestrated within Kubernetes deployments.
3. **DevSecOps integration with Kong microgateway.** The final objective focuses on automation and integration of the Microgateway Infrastructural Pattern with DevSecOps best practices, adopting Kong Gateway as the microgateway technology. Automation servers like Jenkins and GitHub Actions are employed to illustrate the seamless merging of these concepts, emphasizing the importance of adhering to best practices in Full-Lifecycle API Management.

1.2 Structure of the thesis

This thesis document is divided in the following chapters:

- **Chapter 1.** This introduction. It states clearly the objectives of the research and the structure of the thesis.
- **Chapter 2.** This chapter delves into a comprehensive study of the current landscape of **Full-Lifecycle API Management**. It aims at establishing a foundational understanding of the state-of-the-art practices in the API landscape. Concepts like OpenAPI Specification, DevSecOps, Continuous Integration and Continuous Delivery (CI/CD), Infrastructure as Code (IaC) and 42Crunch are presented.
- **Chapter 3.** In this chapter, the **Microgateway Infrastructural Pattern** is introduced as formulated in this research. It encompasses an initial presentation of the adoption of the microservices architecture, along with the strangler pattern and the usage of the hexagonal structure into the microservices context. The Microgateway Infrastructural Pattern is then comprehensively defined, serving as the foundation for the rest of the thesis. Here, **Kong Gateway** is also presented, along with its integration with the tools and methodologies employed throughout the thesis, such as Kubernetes and DevSecOps.

- **Chapter 4.** This chapter extensively explores various enterprise use-cases, all centered around the secure deployment of APIs using Kong microgateways. It delves into **infrastructure configurations** for each scenario, analyzing components integration with Kong microgateways. It includes **authentication and authorization** scenarios with tools like Keycloak and Open Policy Agent (OPA), **monitoring and observability** demonstrations using Prometheus and Grafana, and highlights **secrets management** via HashiCorp Vaults for robust security and access control.
- **Chapter 5.** In this chapter, the focus is on automating the processes within **Continuous Integration and Continuous Delivery (CI/CD)** pipelines, in the context of the Microgateway Infrastructural Pattern implemented with Kong Gateway. The specific emphasis is on leveraging automation tools such as **Jenkins** and **GitHub Actions** to streamline the deployment and testing of the scenarios presented earlier.
- **Chapter 6.** In this chapter, the **validation results** from the research are presented. This involves the combination of functional and performance tests conducted on the use-cases, alongside the integration of 42Crunch analysis and verification of Kong configurations, providing a comprehensive display of the validation process and **its outcomes**.
- **Chapter 7.** The final chapter serves as a summary of the achievements of this research, confirming the successful completion of all the objectives. It also highlights which are possible directions for future work, identifying areas that may benefit from further in-depth study.

2. Full-Lifecycle API Management

In 1970, American computer scientist and software engineer Dr. Winston W. Royce, published "*Managing the Development of Large Software Systems*" [Roy21], the first paper introducing the concepts and presenting the diagram of the **waterfall model**. This model is widely regarded as the first formalized Software Development Life Cycle (SDLC) model. In today's world, businesses are heavily reliant on Application Programming Interfaces (APIs). Just as software development has its lifecycle, APIs follow a structured and systematic series of steps. This journey, which involves both the preparation and exposure of APIs, encompasses the *producer lifecycle* and the *consumer lifecycle*. The holistic management of this entire process is known as **Full-Lifecycle API Management**.

However, as this thesis focuses on the adoption of a **microgateway infrastructural model** to expose APIs on the internet, a comprehensive description of every single step in the lifecycle of every single step in the lifecycle would be out of scope. This chapter is meant to provide detailed information on the most-recent technologies and solutions adopted by the industries. This includes discussions ranging from the concept of **Web APIs** to the top strategies in **full-lifecycle management**. By the end of this chapter, readers will gain a solid understanding of the fundamental concepts and key considerations in the realm of Full-Lifecycle API Management, specifically as it pertains to the adoption of a microgateway infrastructural model for exposing APIs on the internet.

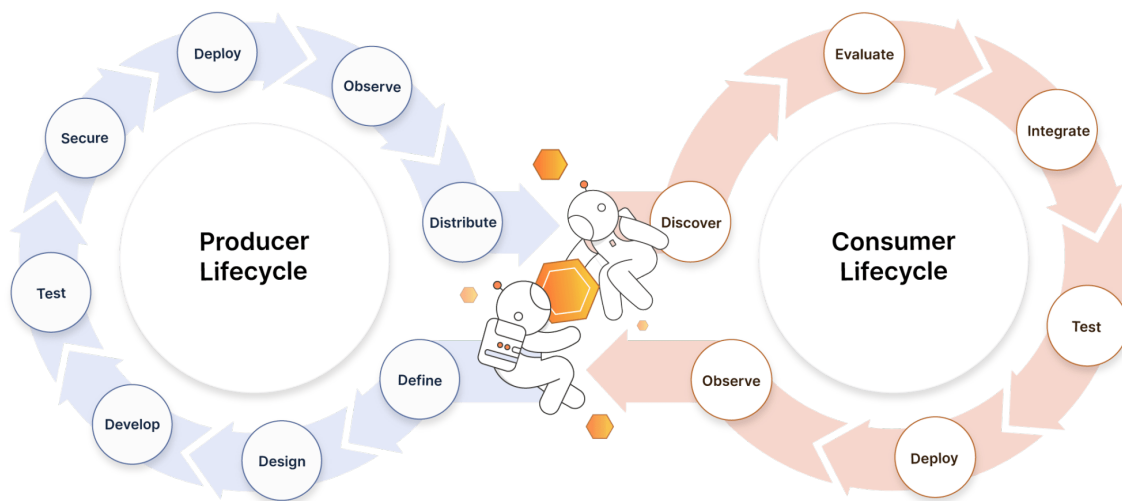


Figure 2.1: Representation of the Full-Lifecycle of the APIs both from the producer and the consumer point of view. [Source: [Postman](#)]

In the upcoming sections, we will explore why Web APIs have become vital assets on the internet and how developers can effectively manage them while incorporating security considerations aligned with the DevSecOps approach. Our focus will be on the integration of the APIOps best practices in a CI/CD scenario for APIs development, including security considerations following the OWASP's API Security guidelines.

2.1 Application Program Interface (API)

The first step requires to understand what APIs truly are. "APIs" stands for **Application Programming Interfaces**, which are generally defined as the *set of rules and methodologies that allow applications to communicate with each others, while hiding the underlying details to their consumers*. Much like a user interface (typically a graphical user interface or GUI) is designed for human interaction with machines, APIs are crafted to enable applications to communicate with one another.

There are many different types of APIs. For instance, the operating system provides a set of APIs that allow an installed application to interact with the filesystem, opening an existing file or creating a new one: the `open()` function in the Unix systems or the `CreateFileA()` in Win32 for Windows are examples of such APIs. Furthermore, various other systems, including social media platforms like Instagram, cloud service providers such as Amazon Web Services, and IoT ecosystems like SmartThings, also define and expose APIs to facilitate interaction with other applications. Consider Meta, the well-known American multinational technology company. Meta exposes APIs over the internet to allow developers writing applications that programmatically query data, create and manage ads, and perform a wide variety of other tasks [Dev23].

2.1.1 A bit of history

Those are common uses for APIs in the world of computing, although APIs, as we understand them today, have a rich and extensive history that predates the advent of general-purpose computers. In fact, it is impossible to discuss APIs without acknowledging their predecessors computer subroutines, which were first introduced in the 1947 book by Goldstine and Neumann [GN47] during their work on the EDSAC computer at Cambridge. However, the term "Application Program Interface" itself made its debut in the 1968 Cotton-Greatorex paper [CG68]. Towards the end of the paper, it is stated:

*"Finally, hardware independence at the central computer means that a consistent **application program interface** could be maintained if that computer were replaced."*

Slowly, from the early IBM instructions set to the Brian Kernighan and Dennis Ritchie's bible: "*The C programming language*", multiple uses and definitions of what constitutes an API have been attempted, leading to the broad usage of the term, as defined by the technologies Carl Malamud, intended as "*a set of services available to a programmer for performing certain tasks*" [Mal90]. In the 1970s, as computer networks became increasingly prevalent, and later in the 1990s, with the widespread adoption of the internet, computers gained the capability to expose services from remote locations. With the emergence of the World Wide Web, protocols and standards were established to facilitate the exposure of APIs over the internet, giving rise to what we now know as Web APIs.

2.2 Understanding Web APIs

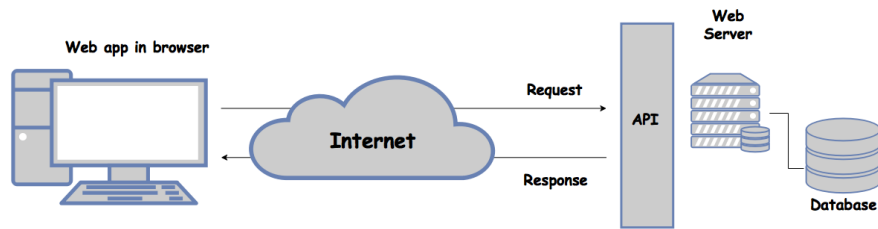


Figure 2.2: Representation of the interaction between a Web Browser and a Web API. [Source: [Analytics Vidhya](#)]

Web APIs are Application Programming Interfaces specifically thought and designed for the interaction between **machines over the World Wide Web**, providing a programmatic access to network-based resources and functionalities. Typically, a client machine (for instance, a Web Browser like Firefox or Chrome) makes a request to a Web server exposing the API, adopting a specific protocol defined in the API specification. The request is sent by the client machine over the internet and it is received and executed by the server. The server may eventually produce a response (the result of some computation or the resource fetched from a database), which is returned to the client machine over the internet again. This flow represents the so-called **Client-Server model**. Although different models exist, what is important here is to understand what Web APIs are: *a piece of code in a remote machine, whose execution is requested (i.e. triggered) from another remote machine over the internet.*

2.2.1 The impact of Web APIs to the modern internet

Growth of API traffic by industry

Top 6 industries by API volume, February to December 2021

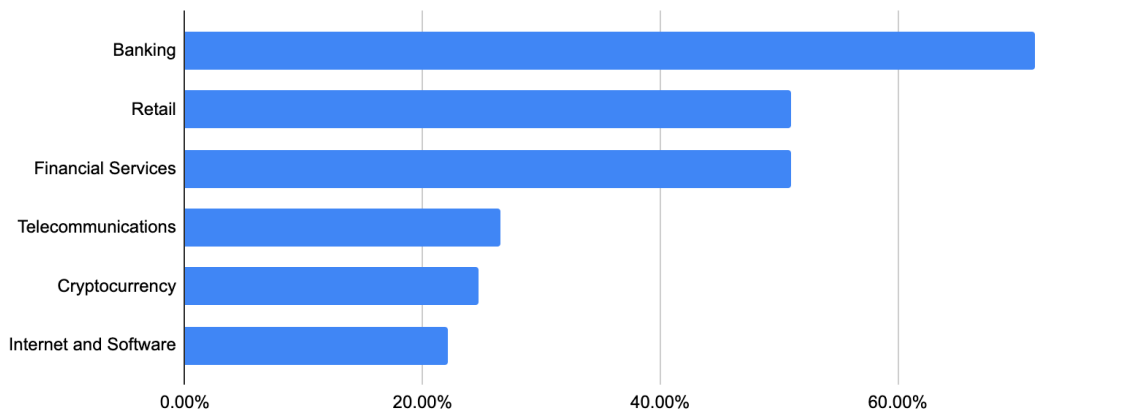


Figure 2.3: Growth percentages of the API traffic per business in 2021. [Source: [Cloudflare](#)]

In 2014, Akamai researches found out that 53% of the Web traffic is content fetched and formatted for humans (mostly HTML or CSS files), indicating that the remaining part accounts for content designed and intended for machines: in other words, API traffic. In 2018, they re-examined the Web, discovering that only 17% of the responses are intended for humans:

APIs accounted for 83% of the Web traffic [Ton18].

Today, APIs play an increasingly vital role in shaping businesses, propelled by a variety of factors. In the last decades, backend servers of websites adopted an API-oriented architecture to create a better developer-experience; the more recent growth of microservices and serverless architectures has further contributed to this trend; IoT and, generally speaking, connected devices such as wearables, sensors and robots are built using an API approach to facilitate inter-communications between different machines. And this is just the tip of the iceberg.

Figure 2.3, from the Cloudflare blog [Mol23], sorts businesses according to their API traffic growth in 2021. For instance, banking, retail and financial services have seen the greatest growth in term of APIs usage. Those are services involved in the exchange of a large amount of money. And where money is the actor, **security** must be taken even more seriously. Indeed, in the same report [Ton18], between May 1 and December 31, Akamai detected 10.000.585.772 credential stuffing attempts solely in the retail industry.

This thesis endeavors to introduce a novel, scalable, and decentralized approach for the exposure of Web APIs, with a focus on ensuring security and safeguarding businesses against data and financial breaches.

2.2.2 How to interact with Web APIs

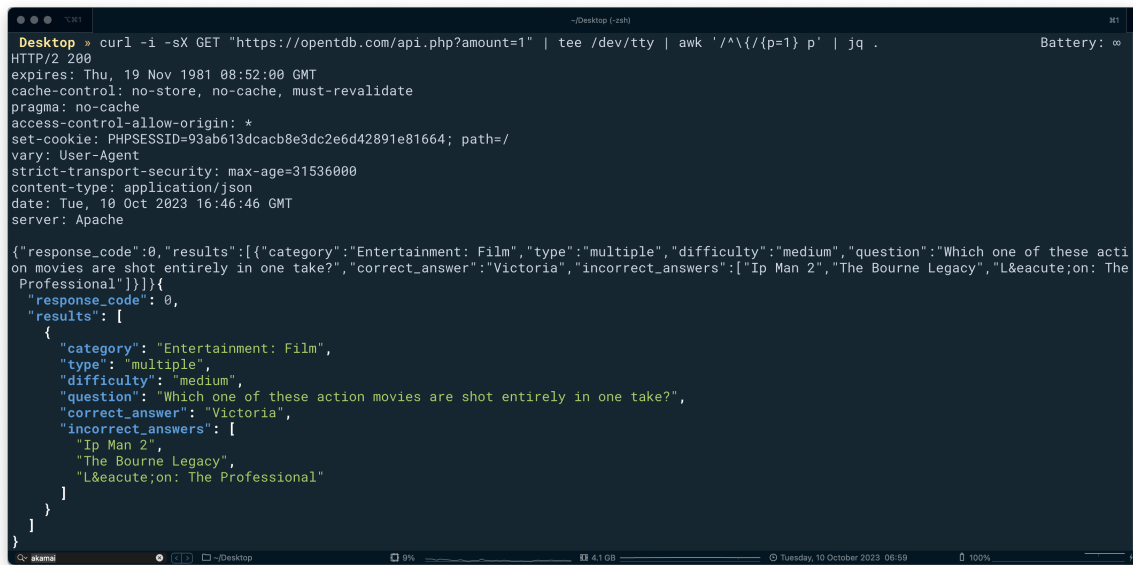
Interacting with a Web API requires knowledge on how the API can be called, where is the API located and what data it can provide. We will see, in the next section, which are the most common protocols and standards adopted by the industries in the Web APIs scenario. However, I will present here an example of request to fetch some data from a Public REST API.

To interact with a Web API, we start by making an API request. This request typically includes various components:

- The **HTTP method** indicates which type of action we want to perform on the API, such as retrieving data, creating new objects, updating entities or removing records from a database. These operations are mapped to the `GET`, `POST`, `PUT/PATCH` and `DELETE` verb of the HTTP protocol.
- The **endpoint** contains information about the location of the API (the Web server that hosts the API) and which resource we want to interact with. It is in the form of a URL (Uniform Resource Locator) like `https://api.example.com/users`.
- The **headers** provide additional information, such as authentication tokens or content types (e.g., JSON or XML).
- Additional **parameters**, like query parameters, permit to filter, sort, or customize the received data.

For the purpose of this demonstration, consider the [Open Trivia Database](#), a Public RESTful service exposing APIs related to trivia questions. The APIs does not require authentication or authorization, therefore we can directly fetch data without any preliminary considerations. We will use the `cURL` command-line tool to interact with the API, attempting to fetch (`GET`) a single random question (`?amount=1`) at the public available endpoint (`https://opentdb.com/api.php`) exposed by the service.

The request is the following:



```
Desktop > curl -i -sX GET "https://opentdb.com/api.php?amount=1" | tee /dev/tty | awk '/^\{p=1\} p' | jq .
HTTP/2 200
expires: Thu, 19 Nov 1981 08:52:00 GMT
cache-control: no-store, no-cache, must-revalidate
pragma: no-cache
access-control-allow-origin: *
set-cookie: PHPSESSID=93ab613dcacb8e3dc2e6d42891e81664; path=/
vary: User-Agent
strict-transport-security: max-age=31536000
content-type: application/json
date: Tue, 10 Oct 2023 16:46:46 GMT
server: Apache

{"response_code":0,"results":[{"category":"Entertainment: Film","type":"multiple","difficulty":"medium","question":"Which one of these action movies are shot entirely in one take?","correct_answer":"Victoria","incorrect_answers":["Ip Man 2","The Bourne Legacy","L\u00e9on: The Professional"]}]}
{"response_code":0,
 "results": [
  {
    "category": "Entertainment: Film",
    "type": "multiple",
    "difficulty": "medium",
    "question": "Which one of these action movies are shot entirely in one take?",
    "correct_answer": "Victoria",
    "incorrect_answers": [
      "Ip Man 2",
      "The Bourne Legacy",
      "L\u00e9on: The Professional"
    ]
  }
 ]
}
```

Figure 2.4: Example of Web API request and response.

From the picture above, we can see that the API interaction happens by using exactly the components previously defined: the method is `GET`, the endpoint is `https://opentdb.com/api.php`, the headers are displayed right after the command is sent to the terminal interface. Specifically, the headers section contains keys like `Content-Type` which indicates the type of accepted response, in this case a JSON value, and more. Finally, the parameters are passed along with the endpoint: `?amount=1`.

The response is, indeed, a JSON-formatted entity, containing the information of the Trivia question. The Web server that manages the API, has received the call from the client and understood the request and which data was necessary to fetch from the database. Finally, it has returned the response to the client. **We have made our first Web API request.** In the rest of this thesis, we will use different and more complex tools, rather than `cURL`, to interact with Web APIs that have been created to complete the objectives.

2.2.3 Protocols and Standards

As seen in the previous section, interacting with a Web API requires developers to have knowledge about a series of specifications that are custom to that service. The increasing adoption of Web APIs in the industry landscape led to the definition of a set of well-defined API-oriented protocols and standards.

2.2.3.1 Types of Web APIs by audience

Three fundamental type of Web APIs are generally recognized by the organizations exposing services to their customers.

1. **Public APIs**, also called **Open APIs**, are application programming interfaces easily accessible with HTTP requests. Those APIs are publicly documented, with well-defined endpoints, versions and schema. They generally require no authentication or easy-to-

grab API keys and are intended for public usages, thus adopting rate-limiting strategies. Example of Public APIs are the [Google Maps API](#), which do require authentication to restrict usages based on the type of subscription.

2. **Private APIs**, also called **Internal APIs**, are designed to be used inside the organization, hidden from the external users. Those APIs intend to improve the functionalities that companies may offer to their internal teams, providing more logging information and load-balancing capabilities.
3. **Partner APIs** are intended for business partners and typically require an on-boarding process in order to obtain credential keys to access the APIs. Those APIs are not publicly available to anyone, but access can typically be requested by the consumers: thus, they exist somewhere in between Public and Private APIs.

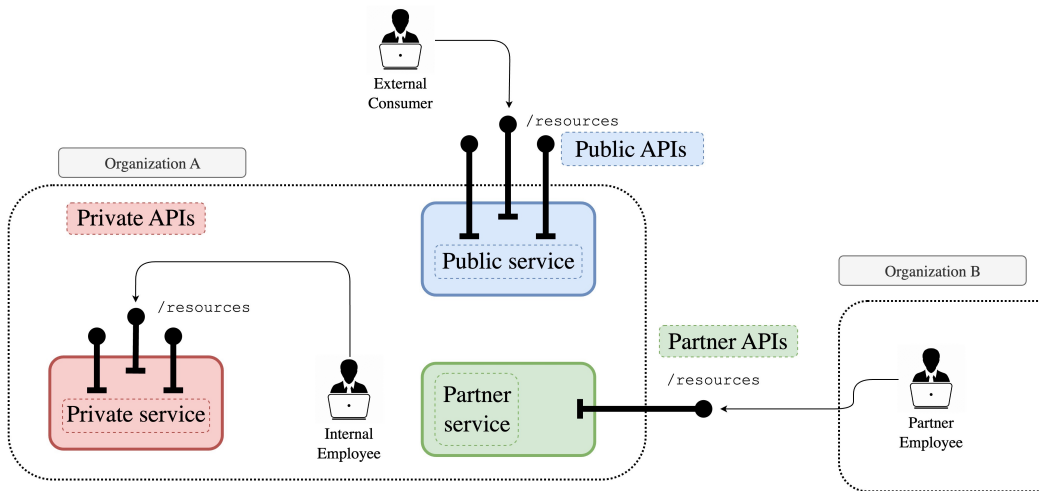


Figure 2.5: Schema representing the different types of Web APIs.

2.2.3.2 Types of Web APIs by architecture

Based on the underlying architecture that companies set up to expose their APIs, we can classify the Web APIs in three categories:

1. **Monolithic APIs** resembles relational databases or MVC applications, offering predictable functionality and stability. However, their interconnected data makes scaling and refactoring challenging, often resulting in concerns about "breaking changes" due to their complexity.
2. **Microservices APIs** represent the primary alternative to a monolithic architecture. Each API serves a specific purpose. Development teams utilizing CI/CD often incorporate multiple microservices in their code lifecycle, ensuring flexibility and modularity, allowing for easy replacement, updates, or retirement of individual microservices without disrupting the entire system. For instance, a logistics company might have microservices for package tracking, route optimization, and driver scheduling within their workflow. **Those are the APIs that are considered in this thesis.**
3. **Composite APIs** enhance data accuracy and minimize data transmission by dispatching a single payload to multiple endpoints. This is a typical trade-off to microservices

APIs, where the required information are sparse over multiple API endpoints.

2.2.3.3 Protocols for Web APIs

To interact with any set of Web APIs, it is necessary to know which are the *means* of the communication, i.e. how it is possible to call a specific API, how to format the request and what to expect as formatted response. We can classify the protocols adopted by APIs developers in four big categories (*note that: although more protocols exist, like gRPC, MQTT or WebSockets, I decided to report here only the most commonly used*).

1. **REST APIs.** Introduced by Roy Fielding ("*Architectural Styles and the Design of Network-based Software Architectures*" [Fie00]), the main designer of HTTP 1.1, REST stands for **REpresentational State Transfer**. REST APIs are based on a typed request/response messaging protocol like HTTP and require a **client-server** architecture, to be **stateless**, **cacheable** and **layered**. Typically, REST APIs work by exchanging JSON-formatted information.
2. **SOAP APIs.** Developed in the late 1990s as a protocol for exchanging structured information in a decentralized, distributed environment, primarily using XML as the message format, SOAP stands for **Simple Object Access Protocol**. SOAP APIs are not limited to HTTP messaging protocol and it uses interfaces instead of simple URL-based resource to expose the API functionalities.
3. **RPC APIs.** Rather than focusing on resources, **Remote Procedure Call** focuses on execution of actions. RPC APIs can work either with JSON and XML, but typically, they do not return a document (i.e. the fetched resource). Publicly exposing RPC APIs is uncommon because most companies are hesitant to grant the general public the ability to invoke methods on remote servers. An example of RPC API is Google RPC. In 2015, Google introduced a type of RPC called gRPC, which uses [Protocol Buffers](#) to serialize and parse data.
4. **GraphQL APIs.** The approach adopted by GraphQL is somewhat similar to REST APIs. However, where REST uses multiple endpoints to locate different resources, GraphQL uses a single endpoint but unlimited data schema available that can be fine-tuned to customize the query in large datasets.

Web APIs is a very large topic, that would require much deeper considerations, time and space to be comprehensively discussed. However, this section aimed at providing a general presentation of the entities that will be needed later on in the thesis. Specifically, this thesis focuses on the secure exposure of REST APIs implemented in RESTful microservice architectures, following the latest trends in APIOps and full-lifecycle management.

2.3 The API Lifecycle

Web APIs run businesses and businesses rely on their APIs to grow. APIs are consumed by clients and partners to create new services for end-users and for these reasons, managing well-established, adaptable and secure processes that produce Web APIs is of pivotal concern for a company. Inadequate and fragile API management solutions expose organizations to several risks:

- They give rise to inefficient and unregulated API development processes, leading to the

creation of inconsistent, poorly documented, and insecure APIs, ultimately resulting in subpar experiences for API consumers.

- They hinder the organization’s ability to tailor API management capabilities to align with evolving business and enterprise architecture goals.
- They offer limited avenues to mitigate solution complexity and control costs.

Gartner, Inc. analyst, Paul Dumas, in the 2023 Report "*Reference Model for API Management Solutions*" [Dum23], wrote:

API management technologies must support the full life cycle of APIs, enable a well-governed API development process, meet distinct organizational needs and be adaptable.

2.3.1 The Producer Lifecycle

As shown in 2.1, a clear distinction is made between two specific lifecycles of APIs:

- The **Producer Lifecycle** refers to those processes related to the design, development and exposure of the APIs on behalf of a particular business.
- The **Consumer Lifecycle**, on the other hand, refers to the processes related to the access, integration and monetization of those APIs for other clients and businesses.

In this section, and in the rest of this thesis, we are going to focus on the **Producer Lifecycle**, understanding which are the phases in which a typical management process is divided in, and which are the most common concerns that developers and operation teams must take into account.

An API Producer Lifecycle may vary from company to company, according to their specific needs. Therefore, although an universal model would not be possible, it is convenient to identify the most common steps that APIs typically traverse before becoming accessible to other clients. In this thesis, eight distinct steps are considered: Define, Design, Develop, Test, Secure, Deploy, Observe and Distribute. Given their interrelated nature, some of these steps are grouped into three primary categories:

- The **blueprinting** processes refer to the early stages of the lifecycle, in which APIs are initially thought and designed.
- The **implementing** processes represent the actual code phase, in which developers implement and test the code that runs the APIs.
- The **exposing** processes relate to the final stages of the lifecycle: APIs are deployed and distributed to consumers under strict monitoring.

Let’s see them in more details.

2.3.1.1 Blueprinting APIs

Stakeholders and Product Managers (PMs) are the main characters of the initial steps in the API lifecycle. Typically, it commences with specific business requirements, subjected to thorough analysis to establish the groundwork for forthcoming endeavors. The following

operations can be considered in the "blueprinting" stage:

1. **Define.** Operational, business and security requirements, along with team members and workspaces are typically defined in this stage. For managers, this step is crucial: a clear separation of concerns, an adaptable but stable plan and long-term product definitions help stabilize the rest of the API lifecycle.
2. **Design.** In the design step, managers clearly defines how an API will expose data to consumers. In large and profitable companies, the design of the APIs is standardized by adhering to organizational patterns and best-practices, in order to shape the continuity of the exposed services. Typically, a set of API specification files is the result of this phase. As we will see in [2.3.2](#), the OpenAPI Initiative aims at defining a language-agnostic interface to HTTP APIs that can be read both by humans and machines [[Sma](#)].

Once the blueprints for the new group of APIs are ready, developers take over the process.

2.3.1.2 Implementing APIs

The central step in the API lifecycle is the implementation of APIs, where developer teams and quality assurance teams collaborate in creating them. The following operations can be considered in the "implementing" stage:

3. **Develop.** Developers start implementing the API in code, typically using a Version Control System (VCS) like Git and a central self-hosted repository in GitHub, GitLab or BitBucket. Clearly, the developers workflow vary widely and it typically happens in cooperation with the QA team to ensure that the API's functionality is thoroughly tested and verified, adhering to project requirements and quality standards. This collaborative effort helps identify and address issues early in the development process, promoting a more efficient and reliable API development cycle.
4. **Test.** APIs testing allows teams to assert that the APIs work as expected. Tests can be automatically managed in a CI/CD scenario, as we will see later, or manually performed. In any case, a multitude of different tests exist, each one suitable for a specific set of assertions (e.g., Unit Tests, Functional Tests or Performance Tests).
5. **Secure.** In this step, the goal is to check for common vulnerabilities that could jeopardize the overall security of an application. These critical security evaluations can be carried out through manual inspections or seamlessly integrated into CI/CD pipelines (as discussed in [2.4.2](#)), making sure that each API within an organization's repertoire upholds consistent and robust security standards.

Once the code of the APIs is validated and ready to serve its consumers, operation teams take over the process.

2.3.1.3 Exposing APIs

Publishing APIs is the final step of the producer lifecycle. With the source code prepared in the central repository, it's time to deploy it to the company's on-premise or on-cloud services. At this juncture, consumers gain access to the APIs. However, for the engineers involved, the work doesn't come to an end here. It remains crucial to continuously monitor the status of the exposed services, ensuring their availability and performance to guarantee a seamless experience for consumers. The following operations can be considered in the "exposing" stage:

6. **Deploy.** In this operation, the meticulously crafted APIs are rolled out into the intended environment, whether it's on-premise or in the cloud. Deployment procedures are executed to make the APIs accessible to consumers, marking a crucial transition from development to active service. In the optimization of deployment processes, organizations often turn to the standardization offered by **CI/CD pipelines** and **API gateways**.
7. **Observe.** After deployment, continuous observation is paramount. Monitoring tools and alerts are employed to keep a vigilant eye on the APIs' performance, availability, and any potential issues. This ongoing monitoring ensures quick responses to disruptions and maintains a high-quality service.
8. **Distribute.** In this final step, the APIs are made available to their intended users. Proper documentation, developer support, and communication channels are established to facilitate access and usage. This distribution phase aims to provide a seamless experience for consumers and promote the adoption of the newly exposed APIs.

These three operations collectively solidify the transition from API development to active usage and maintenance, ensuring that consumers have access to reliable and well-monitored services.

2.3.2 Swagger OpenAPI Specification

The **OpenAPI Specification** (OAS) [[Sma](#)], formerly known as Swagger Specification, is an API format adopted to describe REST APIs through a standard, programming language-agnostic interface. Facilitating the discovery and understanding of a service's capabilities, it empowers both humans and computers without the need for access to source code or any additional documentation. An OAS file is a JSON object that contains all the necessary details to describe the APIs. It can be formatted as JSON or YAML and adheres to a specific and rigorous structure.

The [Official Documentation](#) provides a more detailed resource. In this section, a concise overview of key aspects of the Swagger OpenAPI Specification will be presented, along with the motivations behind the adoption of such tool.

2.3.2.1 Structure of the OAS file

The OpenAPI Specification file follows a well-defined structure that includes information about the API's:

- **paths** : it describes the available endpoints and operations for each path, enabling users to understand how the API can be interacted with.
- **operations** : it contains details of each operation, such as input parameters, data models for requests and responses, and authentication requirements, ensuring clarity for both developers and consumers.
- **components** : it defines reusable elements like schema, security schemes, and responses, promoting consistency and reusability within API descriptions.
- **security** : it specifies the security mechanisms used to protect the API, including authentication methods, token handling, and permissions.

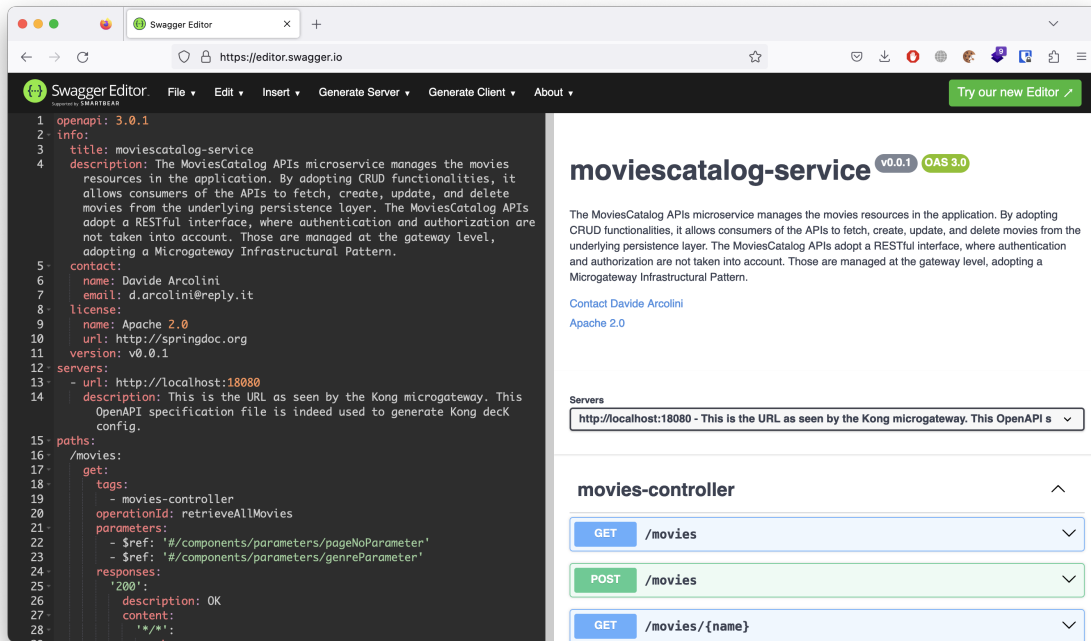


Figure 2.6: A simple CRUD microservice exposing APIs, developed for the purposes of this thesis.

- `info`: it offers general information about the API, including its title, version, contact details, licensing, and terms of use, facilitating consumer comprehension of the API's purpose and terms.

Figure 2.6 represents the OpenAPI Specification file referring to the *MoviesCatalog* microservice developed for the purposes of this thesis (for more details, refer to 4.1). The OAS file has been automatically generated (for more details, refer to 4.1.2.3) in a YAML format, and it contains all the details previously presented.

2.3.2.2 Benefits of adopting OAS

The OpenAPI Specification offers a multitude of advantages. It acts as a linchpin for standardizing the description of APIs, providing a **consistent format** that fosters clarity and uniformity across diverse teams and projects. This standardization ensures that APIs are well-documented, making it easier for teams to work together and understand the intricacies of various APIs without ambiguity. OAS files are designed to be **easily comprehensible** both from humans and machines.

Furthermore, the OpenAPI Specification can be leveraged to create interactive documentation, transforming API descriptions into **user-friendly resources**. Tools like Swagger UI enable users to explore, test, and interact with APIs in a user-friendly and interactive manner. This feature significantly enhances the user experience, making it easier for consumers to engage with and understand the APIs they are working with. Another compelling aspect is its **compatibility** with various development and testing tools. The OpenAPI Specification seamlessly integrates with these tools, streamlining the API development lifecycle. This integration enhances productivity and allows developers to work efficiently, as they can easily incorporate the specification into their existing workflows and processes. As seen in 4.1.2.3,

the integration of OAS with Gradle allowed to automatically generate the documentation of the APIs.

Finally, the OpenAPI Specification is highly adaptable due to its **language-agnostic** nature. It isn't tethered to any specific programming language, which makes it versatile and widely applicable. This versatility allows organizations and developers to utilize the specification across a wide range of programming languages and development environments, ensuring that APIs can be described and interacted with consistently, regardless of the technology stack in use.

2.4 Development, Security and Operations (DevSecOps)

We will operate like developers to make security and compliance available to be consumed as services. We will unlock and unblock new paths to help others see their ideas become a reality.

- DevSecOps Manifesto [Lie]

In the past, software development cycles used to extend over months, and sometimes even years. Within this historical context, the consideration of IT security was typically relegated to the very final stages of the development process. However, the advent of the DevOps paradigm and Agile methodologies has rendered this approach increasingly obsolete. In the realm of modern DevOps, where development cycles can be as short as weeks or even days, outdated security practices can undermine even the most efficiently executed DevOps initiatives.

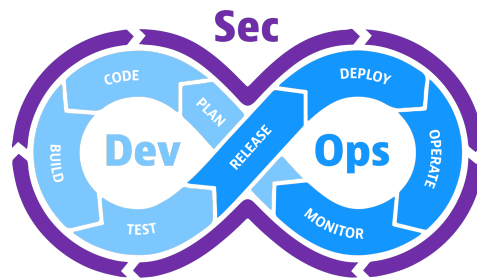


Figure 2.7: DevSecOps cycle. [Source: Dynatrace]

In **DevSecOps** (Development, Security and Operations), security is no longer an isolated concern; instead, it becomes a **shared responsibility** that is seamlessly integrated throughout the entire development process. DevSecOps involves the integration of security within an automated framework that commences at the earliest stages of the software lifecycle. The *Shift Left* "mantra" encourages software engineers to move security from the right-end of the development process, the release of the software, to the left-end, the definition and planning of the software itself (adopting a Security-by-Design approach).

IBM identifies three major benefits from adopting a DevSecOps approach [IBM]:

- **Faster and cheaper software delivery.** Being able to detect and patch security vulnerabilities at the early stages of the development process cuts out duplicate reviews and unnecessary rebuilds, avoiding time-consuming and expensive fixes.
- **Compatibility with automation.** Organizations that employ Continuous Integra-

tion/Continuous Delivery (CI/CD) pipelines for softwares, are able to seamlessly incorporate cybersecurity testing into the automated test suite used by operations teams.

- **Repeatable and adaptive processes.** As the environment changes and new requirements are defined, adopting a DevSecOps approach ensures that security is applied to each phase of the product lifecycle.

In this section, we are going to focus on the API lifecycle and how is the DevSecOps approach applied.

2.4.1 End-To-End Automation with APIOps

DevSecOps is a general paradigm and approach that applies to software development. However, in the recent years, as companies and organizations have shifted towards API-oriented businesses, a new concept has been defined. **APIOps**, standing for API Operations, is built on top of the DevSecOps principles previously presented, and it aims at integrating the security and GitOps best-practices in the development of APIs.

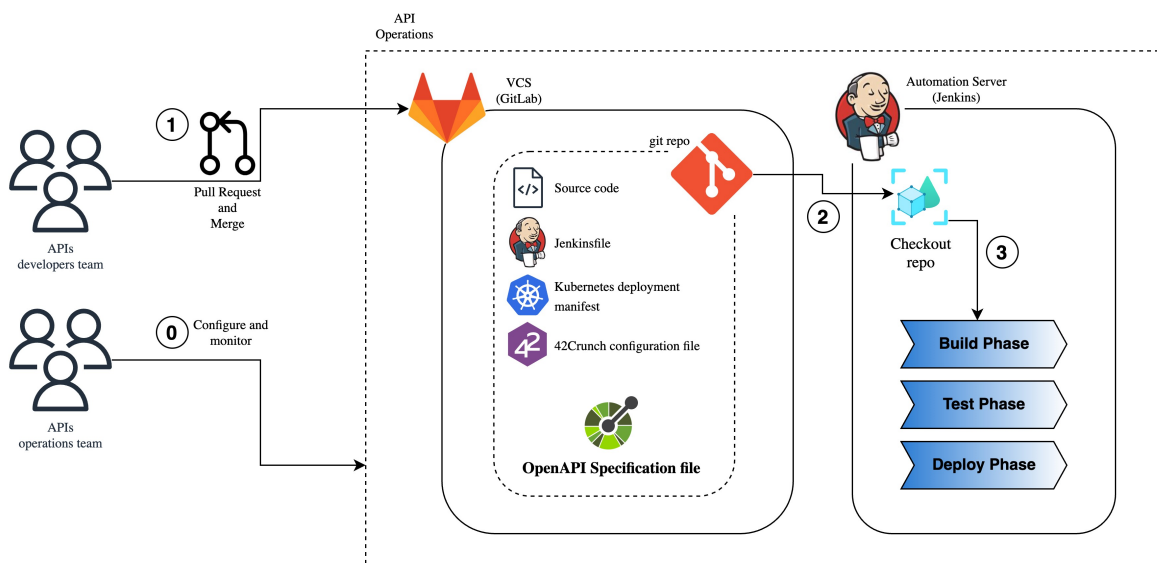


Figure 2.8: Example of APIOps approach using GitLab as VCS and Jenkins as Automation Server.

APIOps is a quite simple paradigm, as it is very similar to the DevSecOps approach. It leverages on the adoption of declarative configurations for CI/CD integration and Infrastructure as Code (IaC) to help developers and operation teams to foster security in the entire API lifecycle. Figure 2.8 depicts a classical workflow in APIOps deliveries.

In the realm of APIOps, the focal point within the Git repository, hosting the source code for the APIs, is the declarative documents outlining the APIs themselves. As this research primarily focuses on REST APIs, the use of an **OpenAPI Specification file** has been adopted. The responsibility of managing this OpenAPI Specification file lies with the developer teams, who manually or automatically generate it and push it into the repository. This file stands as the foundational reference for all other declarative configuration files that come into play during subsequent CI/CD builds. As for the operational aspect of these APIs, the operations

team takes charge by configuring and overseeing the entire pipeline, built around the OpenAPI Specification file. The ultimate goal here is to ensure security at every step of the process, a task that includes interactions with vital components such as 42Crunch servers.

In the upcoming sections, we will delve into detailed presentations of CI/CD, Infrastructure as Code (IaC), and 42Crunch, as these concepts have played a pivotal role in this thesis.

2.4.1.1 Employing CI/CD pipelines

CI/CD, standing for **Continuous Integration (CI)** and **Continuous Delivery (CD)**, is an agile DevSecOps workflow focused on a frequent and reliable software delivery process. CI/CD makes use of automation tools to integrate source code, test it and deliver new releases to the consumers.

Operation	Step	Description
CI	Check-out	The check-out step clones the central repository in the automation server to be used by the agent running the pipeline.
	Test	Unit and integration tests are generally performed at this stage.
	Build	If needed, the source code is built and artifacts are produced.
	Push	The application is dockerized and pushed to a Docker registry.
CD	Deploy	The dockerized application is deployed to test environments (e.g., a K8s cluster specifically designed for testing).
	Test	Functional and performance tests are carried over in this step.
	Review	Deployments are strictly evaluated before reaching the pre-production or production environment. In this stage, results are analysed.

Figure 2.9: CI/CD most common steps to integrate, test and deploy applications.

A CI/CD pipeline is generally composed of:

- An **automation server**, such as Jenkins or GitHub Actions, which controls the agents (i.e., the entities) that run the pipelines. The automation server is typically self-hosted to ensure full control over the confidential information that it has to manage (for instance, API keys to interact with 42Crunch servers).
- A set of **jobs**, or **stages**, which are defined by the operation teams to describe the actions that the agents must perform. The set of operations is generally included in a single file (e.g., a Jenkinsfile or a GitHub Actions workflow), which is set under VCS, and composed of Continuous Integration steps and Continuous Delivery steps.
- A set of **credentials**, keys and confidential information that the agent, running the pipeline, must be in possess of in order to interact with the other components. For instance, consider a CI/CD pipeline that deploys a service in a Kubernetes cluster. The agent must be granted the permissions to interact with the cluster, therefore a configuration file (which should not be disclosed) is stored in the automation server, ready to be used on need.

While several and different steps for a CI/CD pipeline can be defined in according to the developers' needs, in figure 2.9 are reported some of the most common steps that can be found in APIOps automation.

Adopting CI/CD pipelines offers a spectrum of benefits for development teams. Firstly, it

enhances efficiency by **automating testing procedures**, leading to shorter software delivery timelines and enabling **near-instant deployment** of changes to on-premise and on-cloud applications. This efficiency, in turn, translates to a significant reduction in the costs typically associated with traditional software development. Furthermore, the CI/CD pipeline operates as a **continuous loop**, encompassing the stages of building, testing, and deploying code, granting developers immediate feedback to facilitate rapid code enhancements. This early feedback mechanism simplifies debugging processes when issues arise in the initial phases of development. Additionally, the CI/CD pipeline fosters seamless collaboration within the development team, enabling team members to efficiently modify code, respond to feedback, and promptly resolve emerging issues, thus nurturing a unified and agile development environment.

2.4.1.2 Shifting to Infrastructure as Code (IaC)

Infrastructure as Code (IaC) is a transformative approach to managing and provisioning infrastructure in a software-defined and automated manner. It treats infrastructure elements, such as servers, networks, and databases, as code, enabling them to be defined, configured, and deployed through code scripts. Any IaC files should be under VCS just like any other software source code file.

Infrastructure as Code integrates perfectly with the automation required by the DevSecOps approach. For instance, a Kubernetes manifest file can be used to describe the desired state of the APIs deployment in the cluster: the CI/CD agent will be in charge of keep the actual state up-to-date at every build.

Two fundamental approaches to Infrastructure as Code (IaC) can be distinguished: declarative and imperative.

- In the **declarative** approach, the desired system state is defined, outlining the necessary resources and their associated properties. This specification is then used by an IaC tool to configure the system accordingly. Additionally, a record of the current state of system objects is maintained, simplifying the management of infrastructure and changes.
- Conversely, the **imperative** approach revolves around specifying the exact commands required to achieve the desired configuration. These commands must be executed in a specific order. While the declarative approach is primarily adopted by many IaC tools, allowing for the automatic provisioning of desired infrastructure and the implementation of changes when modifications to the desired state occur, imperative tools necessitate determining how to apply these changes.

A declarative approach has been adopted in this thesis.

2.4.2 API Security by Design with 42Crunch

This chapter ends with the description of a security tool that has been used in the development of this research. The details on how the tool has been integrated in the thesis will be presented in Chapter 5.

42Crunch [42C] is a comprehensive platform that specializes in API security, enabling organizations to protect their APIs from potential threats and vulnerabilities throughout the API lifecycle. 42Crunch provides **automated security testing tools** that check APIs for common vulnerabilities, enabling early detection and remediation of potential issues. With its seamless integration into CI/CD pipelines, 42Crunch empowers development and security

teams to collaborate effectively and maintain security as a top priority from the early stages of API development to production deployment.

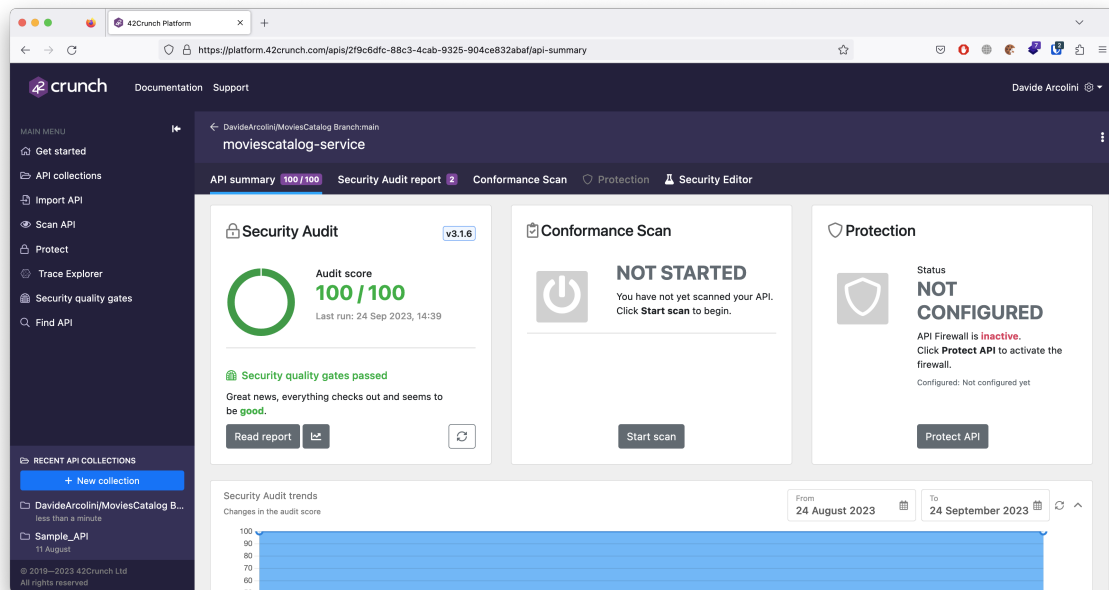


Figure 2.10: 42Crunch interface. [Source: [42Crunch](#)]

In the context of the Continuous Integration and Continuous Delivery (CI/CD) pipeline, the 42Crunch API Audit conducts a **comprehensive analysis of OpenAPI Specification** files. Over 300 checks are executed to enforce best practices and identify potential vulnerabilities in each OAS. The result is an automatically generated report that categorizes issues by severity, covering various criteria such as security, data validation, schemas, and specification format. Additionally, valuable remediation advice is provided within the report to guide the process of issue resolution.

42Crunch is compliant with [OWASP API Security Top 10](#), a project designed to address the increasing deployment of potentially sensitive APIs within the operations of various organizations and deliver value to software developers and security assessors by emphasizing the potential vulnerabilities present in insecure APIs and demonstrating methods to mitigate these risks. To support this objective, the OWASP API Security Project have established and maintained a document [\[Ere23\]](#) outlining the Top 10 API Security Risks, along with a documentation portal presenting best practices for the creation and assessment of APIs.

3. Microgateway Infrastructural Pattern

The previous chapter presented the most recent methodologies and approaches adopted by the industries to manage the APIs lifecycle. As seen in 2.3, securely exposing APIs on the internet is one of the core and fundamental step of the process. This chapter aims at presenting **a new architectural model based on API microgateways**, adopting the innovative and commercial product **Kong Gateway**, to facilitate developers and operation teams in managing company’s services and securing exposure of the APIs.

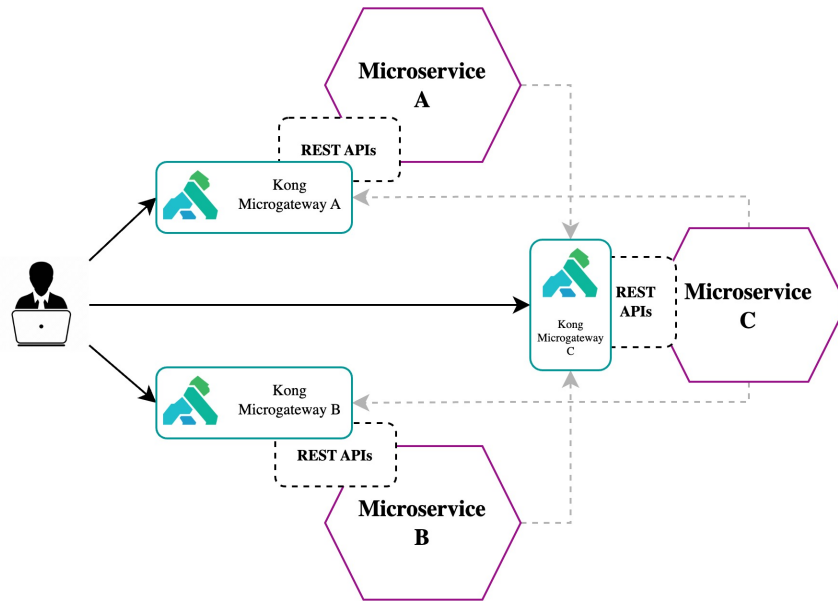


Figure 3.1: Abstraction overview of a general architecture developed adopting a microgateway infrastructural model with Kong Gateway.

In response to today’s intricate and ever-changing business landscape, organizations have found it necessary to transition their IT services to an architectural paradigm that enables **faster and more reliable product delivery**. This imperative has led to the organization of teams into smaller, loosely coupled, cross-functional units. The **microservices paradigm** emphasizes the continuous delivery of changes within specific subdomains, facilitated by the adoption of DevOps principles and the automation of deployment pipelines. However, even as the architectural definition of APIs evolved towards a microservices approach, the prevailing method for exposing APIs remained the centralized and monolithic API gateway.

Within this chapter, a comprehensive analysis aims to shed light on both the strengths and

limitations associated with the adoption of this conventional approach within the dynamic microservices landscape. Subsequently, the innovative **microgateway model** is presented, aligning with the overarching trend of decoupling and decentralizing services and organizations. The final section explores the features of Kong Gateway and it presents the implementations of the proof of concepts developed during this study. Kong represents an innovative player in the realm of API management, and adapting it to a microgateways framework required extensive testing and validation of various architectural configurations. It's worth noting that this section primarily focuses on the **theoretical analysis** of the model. Upcoming chapters will enhance the understanding by showcasing real-case scenarios in which Kong is seamlessly configured to interact with other integral components.

3.1 The rise of microservices

"Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure."

- Melvin Conway, 1967 (*Datamation* [Con68])

In recent years, the term **microservices architecture** has captured the imagination of academics, business researchers, developers, and engineers alike, sparking fervent discussions and garnering substantial attention. Yet, it's worth noting that the roots of this approach extend far back in time.

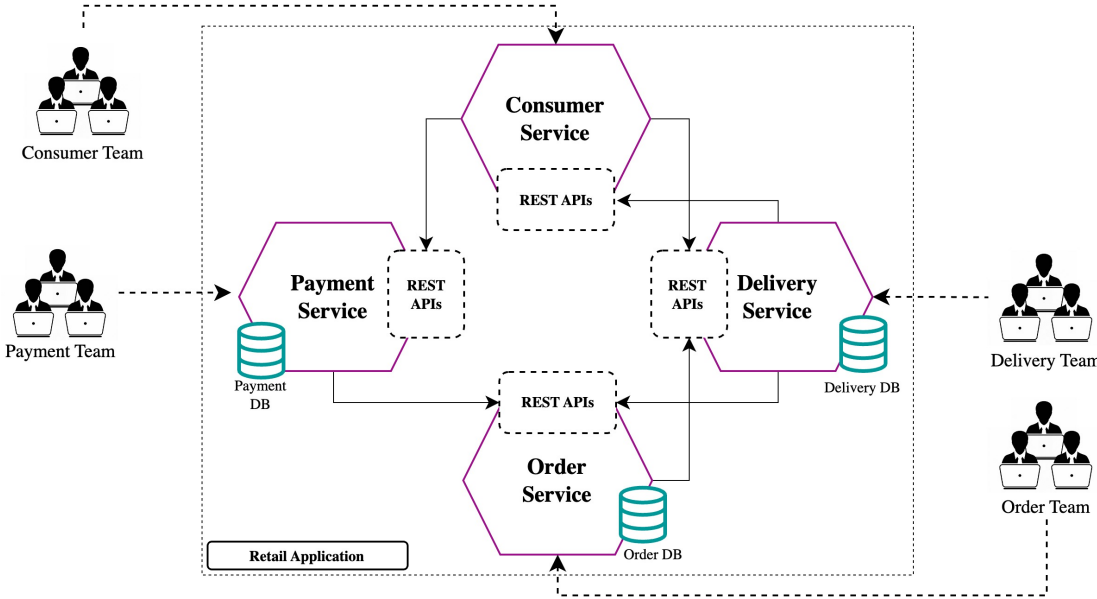


Figure 3.2: General architecture of a retail application adopting a microservices architecture model.

A microservices architecture represents a software architectural approach in which substantial applications are composed of discrete, **self-contained modules** that collaborate via language-agnostic APIs. Each microservice serves a single, small, and meaningful functional feature that the organization provides to their customers. It has its own process and it communicates with other microservices to produce the attended result. Each microservice has a limited scope and

it concentrates on a specific task. Therefore, well-designed interfaces that allow microservices to interact with one another is even more crucial to the development of an efficient application. While there is no universally agreed-upon definition of microservices, the following list outlines the key characteristics that a service must exhibit to be categorized as a **microservice**:

- **Independently deployable**: a microservice should be capable of independent deployment. To achieve this, it must offer a well-defined set of APIs encompassing clear endpoints, data types, relationships, and more. These APIs enable seamless integration into diverse systems while maintaining the ability to interact with other components effectively.
- **Autonomously management of data persistence**: each microservice should maintain its own dedicated database for data persistence. This feature facilitates service replication when necessary, further promoting independence and reducing dependencies on other components.
- **Scalability**: microservices should be designed to scale horizontally, enabling effortless replication and distribution to accommodate varying workloads.
- **Resilience**: microservices should be robust and resilient. They should be able to withstand failures, recover gracefully, and not disrupt the operation of the overall system. Typically, resilience of the system is accomplished by isolating microservices from each other, ensuring that a failure in one service does not adversely affect others.
- **Highly maintainability and testing**: each microservice should be managed by a small, dedicated team performing fast development cycles and easy deployment adopting best-practices from DevSecOps and CI/CD approaches.

Due to these features, microservices architectures are frequently embraced in various technological contexts, including cloud-native applications, serverless computing, and applications utilizing lightweight container deployment. In the realm of **cloud-native applications**, microservices enable organizations to leverage the dynamic and elastic nature of cloud environments, allowing them to efficiently adapt to changing workloads and resource demands. In the context of **serverless computing**, microservices provide a modular and efficient way to build and deploy individual functions, aligning perfectly with the event-driven, stateless model of serverless platforms. Moreover, when applied to **lightweight container deployment**, microservices contribute to improved resource utilization and agility, allowing for the efficient packaging, distribution, and orchestration of individual components within containers.

3.1.1 A bit of history

Melvin Conway, in the 1968 article published in *Datamation* [Con68], asserted that the structure of a system's design mirrors the communication structure of the organization responsible for that same design. He wanted to stress the fact that there is a **deep correlation** between the underlying set of people and the result produced. What he wrote served as the foundational concept upon which microservices architecture is built: in a microservices architecture, the separation of concerns for each service must mirror the division of teams. The evolution of this architectural paradigm, spanning from its inception to its contemporary prominence, reflects an intricate journey marked by innovation, adaptation, and a profound impact on the world of software development.

The term "*Micro-Web-Services*" made its debut in a presentation at the 2005 Cloud Computing

Expo by **Dr. Peter Rodgers**. Following the trend on **Service-Oriented Architectures** (SOA), he suggested that "*any service, at any granularity, can be exposed*" [Rod05]. Nevertheless, the term **microservices** first emerged in May 2011 during a workshop of software architects in Venice, to indicate a common architectural pattern that all of them saw and studied recently. It was in May 2012 that the same group reached a consensus on "microservices" as the most fitting name for this architectural approach [Nic17].

In February 2020, the Cloud Microservices Market Research Report forecasted that the global microservice architecture market would experience a substantial growth, with an anticipated Compound Annual Growth Rate (CAGR) of 21.37% from 2019 to 2026, ultimately reaching a market size of \$3.1 billion by 2026 [Res20].

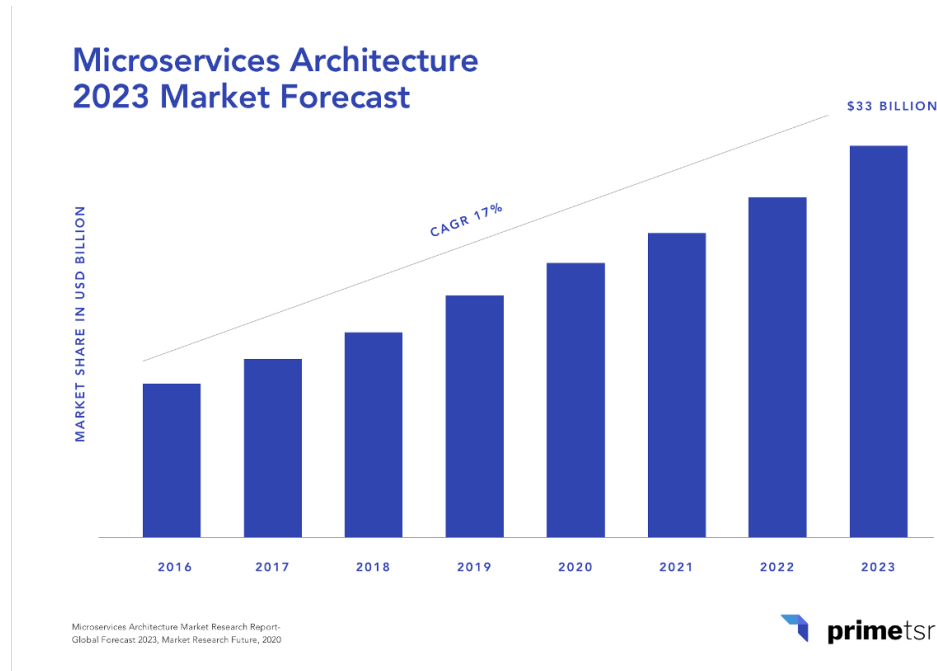


Figure 3.3: Microservices architecture forecast market growth by Market Research at 2023. [Source: Market Research]

3.1.2 From monolith to microservices

Historically, software applications have adhered to the traditional **monolithic architecture**, where all the constituent components were intricately combined into a single, unified entity. This architectural paradigm had its roots in the early days of computing and application development. In fact, in this early stages of software engineering, applications were often built as single, all-encompassing program. These monolithic applications bundled the user interface, business logic, and data management into a single, tightly-knit structure. It was a pragmatic approach that suited the technology constraints of the time.

Consider now the classic three-tier architecture shown in 3.4, that serves as a hallmark of many web applications. The front-end component, responsible for user interaction and presentation, is tightly coupled with the back-end, where the core business logic reside and interface with the data layer. This data layer is dedicated to managing the storage and retrieval of data within the application. Monolithic applications are typically deployed on a set of identical servers behind a load balancer. This offers a straightforward approach to web applications

development, which fits pretty well for **small applications**:

- Applications are **simple to develop**. With the entire business logic residing within the same process, engineers do not need to worry about fetching data from other services. This cohesive structure simplifies the development process.
- Applications are **simple to test**. End-to-End testing can be seamlessly conducted by deploying the complete application in a dedicated test environment and executing the required tests.
- Applications are **simple to deploy**. The monolithic architecture simplifies deployment since the entire application is bundled into a single unit, making it easier to manage and distribute.
- Applications are **simple to scale horizontally**. Scaling horizontally when necessary, is relatively straightforward, as you can replicate the entire monolithic application across multiple servers behind the load balancer to accommodate increased workloads.

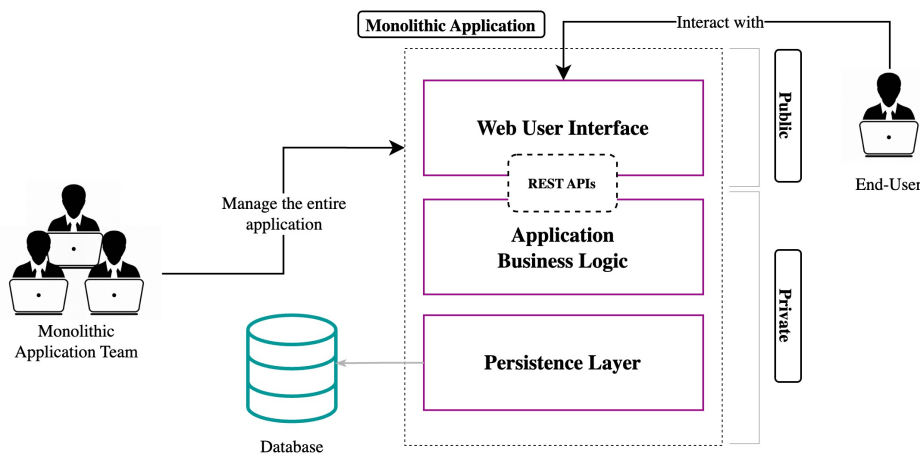
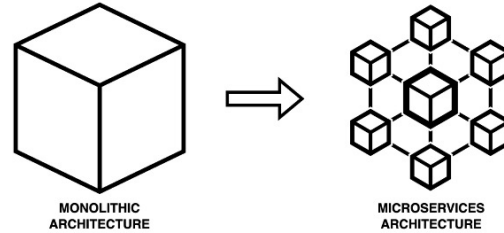


Figure 3.4: General architecture of a monolithic application.

However, the monolithic design comes with some inherent limitations. As applications **grow in complexity and scale**, monolithic architectures poses challenges in terms of maintainability, scalability, and flexibility. Testing the application becomes a serious issue, as well re-deploying the entire monolith takes time and money. Over time, development teams face the dual challenges of technology stack obsolescence, alongside the increasing maturity of system designers necessitating system rewrites, although resource constraints often make it unfeasible. Concurrently, market demands require swift, frequent, and dependable innovations in the product. As we have seen in at the beginning of this section, one potential solution to this conundrum lies in transitioning the system into a microservice-based architecture.

The path to establishing a microservice infrastructure begins with application requirements, leading to three key steps. First, **system operations** are identified through user stories that detail user interactions. Second, the system is broken down into **services organized around business concepts**. Lastly, the third step entails defining each service's **API**, with some services implementing operations independently and others relying on support from fellow services. However, in scenarios where existing monolithic applications are already in place

and starting from scratch is neither possible nor convenient, the **strangler pattern** proves valuable.



3.1.2.1 The strangler pattern

The **strangler pattern** (coined by Martin Flower in 2004 [Fow04]) is a strategic approach employed in web applications architecture to gradually migrate from a monolithic system to a microservices one. In situations where starting from scratch is neither feasible nor practical due to existing monolithic applications, the strangler pattern offers an effective solution.

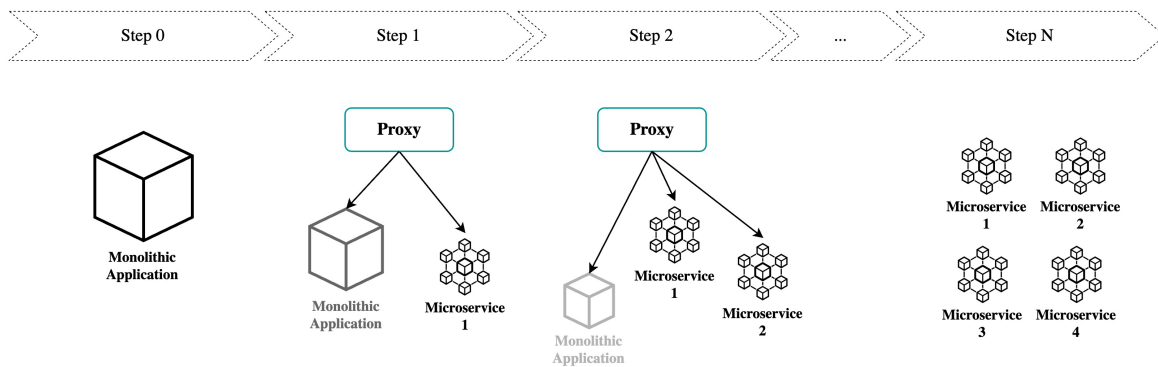


Figure 3.5: Strangler Pattern allows monolithic applications to shrink down to microservices in a controlled environment.

The process begins by creating a new microservices application, which initially implements a subset of the original system’s functionalities. Both the new and old applications share the same URI space to maintain a consistent user experience. As the transition unfolds, the original application is systematically modified to redirect (here the proxy) specific incoming requests to the new one, allowing the two systems to coexist harmoniously. Over time, the new microservices application is incrementally enhanced, progressively taking over additional functionalities from the old system until it ultimately replaces it entirely. The strangler pattern enables a smooth and controlled migration to a more flexible and scalable microservices architecture while preserving the stability and continuity of the existing application.

3.1.2.2 Hexagonal architecture

The **hexagonal architecture**, also known as the ports and adapters architecture, is a software design pattern used to create a system with loosely coupled application components. These components can be seamlessly connected to their software environment through the use of ports and adapters, enabling interchangeability at various levels and streamlining the process of test automation.

Alistair Cockburn introduced the hexagonal architecture in 2005 to mitigate well-documented structural challenges in object-oriented software design. The primary objectives were to **prevent undesired dependencies between layers** and the mixture of user interface code with business logic [Ste14].

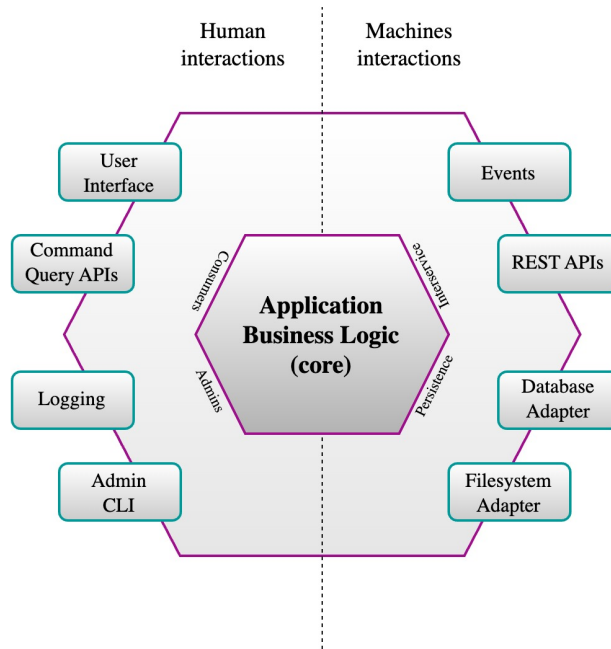


Figure 3.6: Hexagonal software architecture adapted to a microservice implementation.

Although the origins of the hexagonal architecture are rooted in the software design, some authors consider it a valuable design for microservices. In figure 3.6, I want to show how the design can also work for a microservice implementation. Specifically, the hexagon related to a microservices architecture design should be composed of, at least, the following "ports", in order to be able to easily interconnect with other microservices:

- **Consumers** ports allow for the interaction between the service itself and external APIs consumers. If that is the case, a User Interface (UI) may be exposed to facilitate this interaction.
- **Admins** ports allow administrators to configure, manage and interact directly with the microservice, exploiting admin privileges to evaluate logs and query commands.
- **Interservice** ports allow the inter-communication between microservices. Typically, this involves the creation of REST APIs and Events subscriptions to link different hexagons together.
- **Persistence** ports allow the interaction between the microservice and the underlying system. Typically these interactions involve database queries or access to the server filesystem.

Finally, the **business logic kernel** serves as the core of the service, responsible for implementing API operations and event publication. It consumes other services' APIs and may

subscribe to their events. Maintaining the kernel's independence from dependencies is crucial, achieved through adapters that handle external connections (e.g., DBMS, message brokers, APIs).

3.1.2.3 Benefits and drawbacks

The microservices architecture introduces **additional complexity** and novel challenges which should be addressed at scale: network latency, message format design, backup, availability, consistency, load balancing, and fault tolerance. Moreover, the operational complexity may replace some of the complexity of a monolithic application: for instance, increased network traffic can lead to slower performances and applications composed of multiple microservices have a higher number of interface points, elevating architectural complexity.

However, adopting a microservices architecture has several advantages. Here some of the most common are reported, though each different use-case can present variations and additions to this list:

- **Modularity.** In a microservices architecture the application's comprehensibility is enhanced. The separation of domains helps teams clearly define each concern, development process, testing, and resistance to architectural degradation. This benefit is often emphasized in contrast to the intricacies of monolithic architectures.
- **Scalability.** Microservices, operating independently within separate processes, allow for individual monitoring and scaling. When change is required in a certain part of the application, only the related service can be modified and redeployed: there is no need to modify and redeploy the entire application. Moreover, integration with third-party services is easier.
- **Fault isolation.** In monolithic architecture, one problematic area can jeopardize the entire system. Microservices try to maximize isolation to prevent down statuses and develop a fault-tolerant system.
- **Modernization of legacy systems.** Microservices present a viable path to modernize existing monolithic software applications. Numerous companies have reported successful replacements or partial replacements of their legacy systems with microservices. This modernization process adopts an incremental approach.
- **Distributed Development.** Microservices parallelize development by enabling small autonomous teams to independently develop, deploy, and scale their respective services. It also promotes the emergence of an individual service's architecture through continuous refactoring. Microservice-based architectures facilitate continuous integration, continuous delivery, and deployment in distributed environments.

To maximize the benefits coming from the adoption of such architecture, several technologies have been developed. In the next paragraphs, we will see how to manage microservices with Kubernetes (K8s).

3.1.3 Orchestrate microservices with Kubernetes

"Kubernetes, also known as K8s, is an open-source system for automating deployment, scaling, and management of containerized applications." [23e]

Kubernetes, often abbreviated as K8s, is a complex and fascinating tool. It is the cornerstone of the infrastructure for this study, although a comprehensive description exceeds the study's scope. Within the realm of this thesis, Kubernetes serves as the fundamental layer upon which the entire proof of concept is built. The objective of this section is to explore the relationship between Kubernetes and the microservices architecture, shedding light on its indispensability as a tool integrated into the **microgateway infrastructural pattern**. This exploration aims to underscore the essential role Kubernetes plays in this context, offering insights into its pivotal position within the microservices ecosystem.

The upcoming chapters will present, with more details, each single configuration of the Kubernetes cluster adopted in this thesis.

3.1.3.1 Introduction to K8s

Kubernetes is a **container orchestration tool** designed to automate the deployment, management, and scaling of applications. Originally developed by Google engineers and open-sourced in 2014, Kubernetes has evolved into a versatile computing platform and ecosystem, challenging, and in some cases surpassing, virtual machines (VMs) as the fundamental building blocks of modern cloud infrastructure and applications. This ecosystem empowers organizations to deliver a highly productive Platform-as-a-Service (PaaS) solution, effectively addressing numerous infrastructure and operational challenges associated with cloud-native development, allowing development teams to concentrate on coding and innovation.

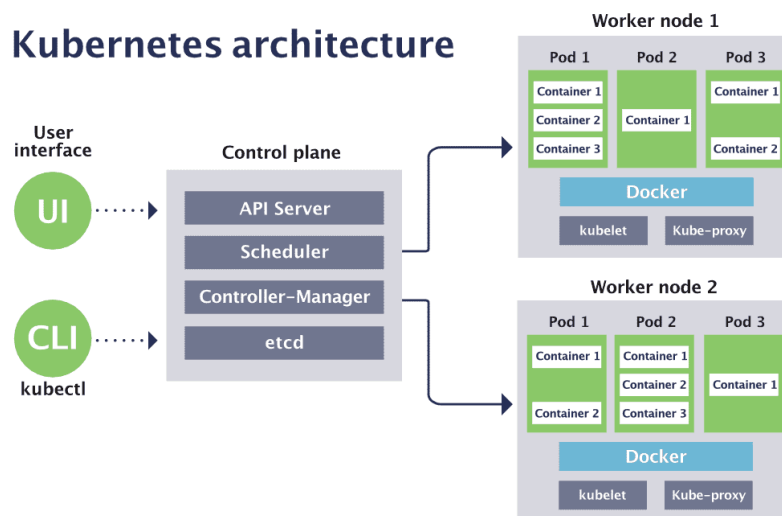


Figure 3.7: Kubernetes architecture [Source: [Cloud Native Computing Foundation](#)].

Kubernetes automates container-related tasks throughout the application lifecycle, including deployment, rollouts, service discovery, storage provisioning, load balancing, autoscaling, and self-healing. It deploys containers to specified hosts, maintains their desired state, manages changes in rollouts, provides service discovery through DNS or IP addresses, provisions storage, balances loads for performance, scales automatically in response to traffic spikes, and ensures high availability by restarting or replacing failed containers and enforcing health-check requirements.

3.1.3.2 Managing microservices concerns

Integrating Kubernetes into the microservices ecosystem addresses several critical concerns and challenges that come with managing the complexity of the architectural design. Kubernetes provides a robust solution for orchestration, ensuring efficiency and reliability in various aspects of the deployment and management process.

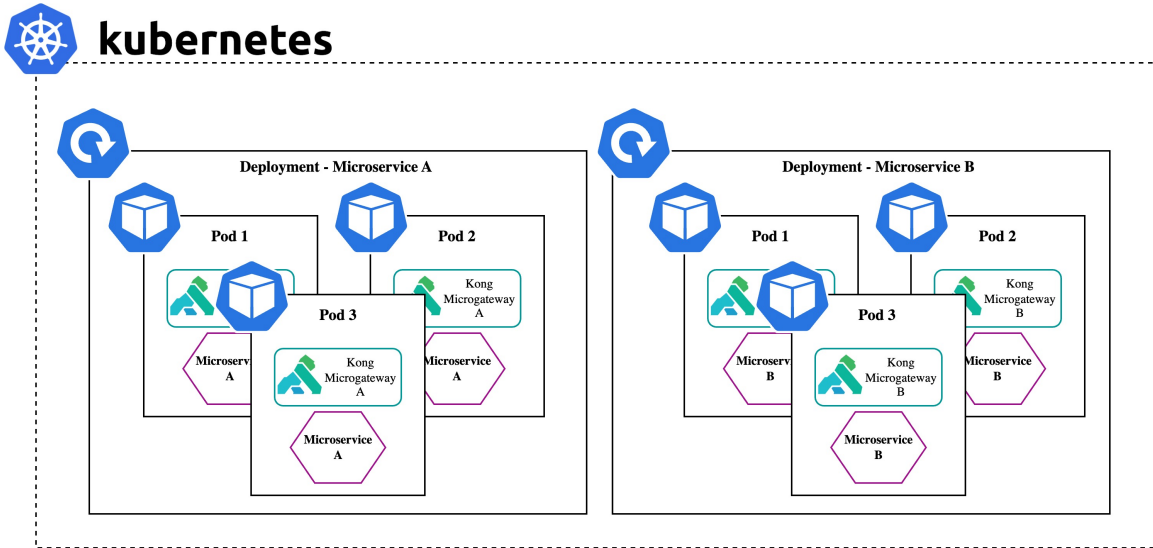


Figure 3.8: Example of Kubernetes cluster used to manage the deployments of two microservices, each of which has 3 replicas deployed as pod. Microgateways are deployed close to the microservices, inside the same pod. The picture does not comprehend all the other entities involved in the development of this thesis (such as: K8s services, Databases, Vaults, IdP, etc...).

- **Efficient Deployment.** Kubernetes streamlines the deployment of microservices by orchestrating the allocation and scaling of containers as needed. This efficiency is crucial in ensuring that microservices are readily available to meet application demands.
- **Scaling and Load Balancing.** The platform's auto-scaling capabilities automatically adjust the number of container instances based on traffic and resource requirements, ensuring that microservices can handle varying workloads effectively. Load balancing further optimizes performance by distributing traffic intelligently.
- **Service Discovery.** Kubernetes simplifies service discovery for microservices by automatically exposing them to the network via DNS names or IP addresses. This simplifies inter-service communication and connectivity.
- **Rollouts and updates.** Managing rollouts and updates of microservices is simplified with Kubernetes. The platform provides features for pausing, resuming, or rolling back rollouts, ensuring a smooth transition during updates or changes.
- **Self-healing.** Kubernetes enhances the reliability of microservices by automatically detecting and recovering from container failures. It can restart or replace failed containers, minimizing downtime and ensuring high availability.

Figure 3.8 shows an PoC adaptation of the initial figure of this chapter, resembling the ad-

vantages of adopting Kubernetes as a microservices orchestrator.

3.2 Exposing APIs on the internet

In the preceding section, we delved into the microservices architecture, comprehensively exploring its principles, advantages, and the rationale behind transitioning from a monolithic design to this more flexible and scalable approach. We discussed the methodologies that guide this migration, emphasizing the value of modularization and decentralized components for enhancing the development and maintenance of web applications systems.

Now, in this section, we shift our focus to the **microgateway infrastructural pattern**. This pattern extends the principles and benefits of microservices to the domain of **APIs exposure and management**. We will examine how the same fundamental concepts that underpin the microservices architecture are equally applicable when it comes to exposing and orchestrating APIs in a manner that enhances agility, security, and flexibility. As we delve into the microgateway infrastructural pattern, we'll elucidate the critical role it plays in facilitating streamlined API management within a microservices ecosystem, while harnessing the same principles that make microservices a compelling approach for modern software development.

3.2.1 The traditional approaches

In the context of **exposing APIs on the internet**, the traditional approach often involves the utilization of API gateways. An **API gateway** acts as a mediator between client applications and backend services in microservices architecture, functioning as a software layer that serves as a **single endpoint for various APIs**. Its role encompasses tasks such as request composition, routing, and protocol translation, enabling the management of traffic and enforcing security policies for APIs.

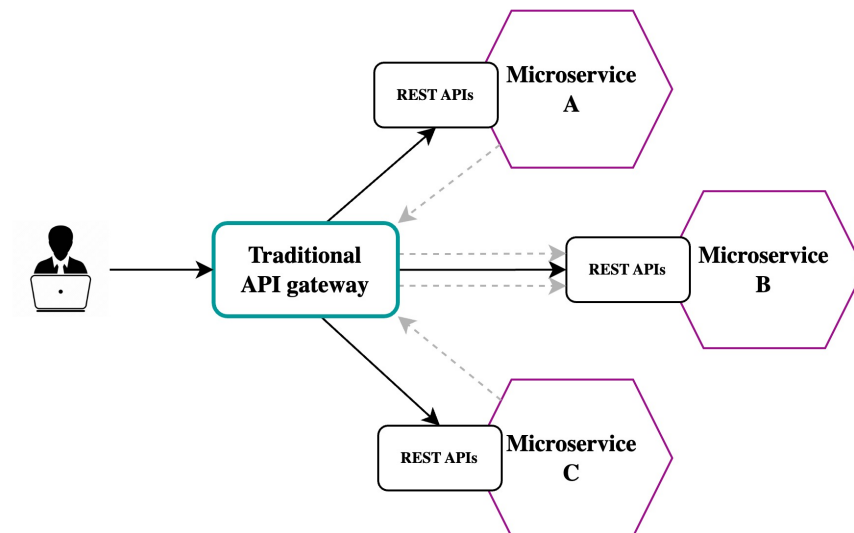


Figure 3.9: Traditional and monolithic API gateway proxying all the requests to the correspondent microservice.

Before we get into the specifics of the API gateway approach, let's understand the contrast between this method and the direct communication between clients and microservices.

3.2.1.1 Direct client-to-microservices communication

An alternative approach for exposing APIs within a microservices infrastructure is **direct client-to-microservice communication**. In this setup, client applications send requests directly to specific microservices, each of which typically features a public endpoint (for example, a simple microservice deployed as an Azure Container Instance (ACI) might have an endpoint like `http://eshoponcontainers.westeu.cloudapp.azure.com:88/` [Mic22]).

While this approach can function effectively for applications composed of a small number of microservices, it warrants careful consideration, as it lacks the use of an API gateway for internet API exposure.

1. Microservices must now **manage the entire request**, which includes implementing logic for policy enforcement, data transformation, protocol translation, and other tasks. This deviates from the core tenets of microservices, which ideally focus on minimal functionality, abstracting away **supporting subdomains** like authentication or authorization, which can be specific to each use case.
2. Without an API gateway, the burden of **security and access control** is shifted to individual microservices. This means that each microservice must independently handle concerns like authentication and authorization, potentially resulting in duplicated efforts and increased complexity.
3. Scalability becomes less straightforward in this model. With no centralized gateway to manage traffic and load balancing, **each microservice needs to handle its scaling independently**, which can be challenging to coordinate in large, complex applications.
4. **Monitoring and analytics can become fragmented** when dealing with direct client-to-microservice communication. Without a centralized point for data collection and analysis, tracking the performance and behavior of the system may require a more complex setup.
5. Maintenance and updates may present additional challenges, as changes to common functionalities, like security policies, may necessitate **modifications across multiple microservices**, increasing the potential for errors and inconsistencies.

In summary, the approach of direct client-to-microservice communication provides a straightforward means of interaction, particularly for applications comprising a limited number of microservices. The subsequent sections will delve into an alternative pattern, the **API Gateway Pattern**, which addresses these challenges by introducing a centralized, intermediary layer to manage API exposure and optimize various aspects of microservices communication.

3.2.1.2 API Gateway Pattern

If direct client-to-microservice communication fits well for small applications, scaling up the number of microservices requires to perform additional considerations, which are efficiently summarized by Richardson in *"Microservices Pattern"* [Ric18]. Microservices often offer fine-grained APIs, requiring clients to interact with multiple services, each tailored to specific data needs. Various clients may require distinct data sets, such as more elaborate desktop versions compared to mobile. Different client types experience varying network performance, with mobile networks typically slower and higher latency. This results in performance discrepancies between native mobile clients and server-side web applications. Additionally, the dynamic

changes in service instances and their locations, evolving partitioning, and diverse protocol usage must be abstracted from clients, presenting a challenge in microservices architecture design.

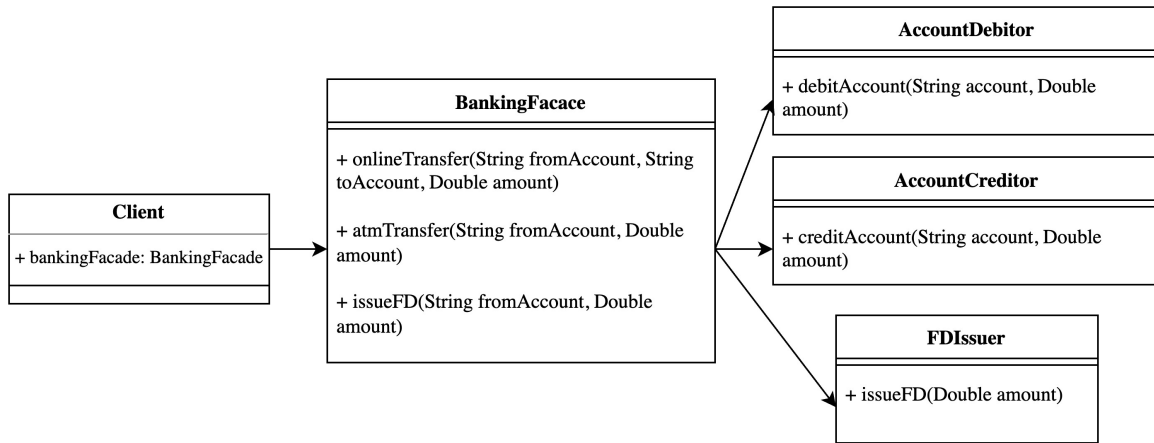


Figure 3.10: Simplified example of a banking system implementing a **Facade Pattern** to manage the banking subsystems. The similarities between a facade and an API gateway can be easily seen in the diagram.

For these reasons, when designing a large application with multiple microservices, the **API Gateway Pattern** is generally a good approach to manage all the concerns previously presented. This pattern shares similarities with the **facade pattern** (see figure 3.10) in object-oriented design but serves as a reverse proxy within distributed systems. The **API gateway** encapsulates the underlying system architecture and it provides clients with a single entry-point to access the whole set (or a subset) of the APIs, acting as a reverse proxy to route the requests.

The API Gateway pattern offers numerous advantages. By managing client requests and routing them to internal microservices, it is possible to incorporate several valuable features within the API gateway by managing cross-cutting functionalities.

- **Authentication and Authorization** can be managed at the gateway level. Enforcing policies and validating client identities ensure that each microservice can focus on the specific subdomain it was designed for.
- **Response caching** allows the API gateway to store in memory responses from microservices to improve performance and reduce redundant requests.
- **Service discovery** allows to dynamically locate and route requests to available microservices.
- **Rate Limiting and Throttling** controls the rate at which requests are processed, preventing abuse or overloading of microservices.
- **Load Balancing** serves the purpose of evenly distributing incoming requests among various instances of a microservice. This strategy aims to enhance resource utilization and bolster fault tolerance.

- **Logging, Tracing and Correlation** help in monitoring and debugging distributed systems. Correlation ensures that requests and responses across microservices are linked.

However, API gateways are typically designed as monolithic entities. While they offer a centralized solution for managing various cross-cutting concerns, this monolithic nature can introduce its own set of challenges, as we have seen in 3.1. As the application grows and evolves, maintaining and scaling the API gateway can become complex. Changes or updates in one part of the system might require modifications to the entire gateway, impacting the agility of the development process.

3.2.2 A new approach adopting microgateways

While API gateways have played a pivotal role in API management, their traditional design often exhibits a **monolithic nature**, resembling tightly integrated systems. This monolithic structure can limit agility and adaptability to the evolving needs of modern applications, as well as introduce potential points of failure. Additionally, traditional API gateways may present challenges in terms of scalability and customization, potentially adding latency to API requests. These limitations set the stage for exploring the **microgateway infrastructural pattern**, which aligns more closely with the principles of microservices and offers a more agile, decentralized, and flexible approach to API exposure and management.

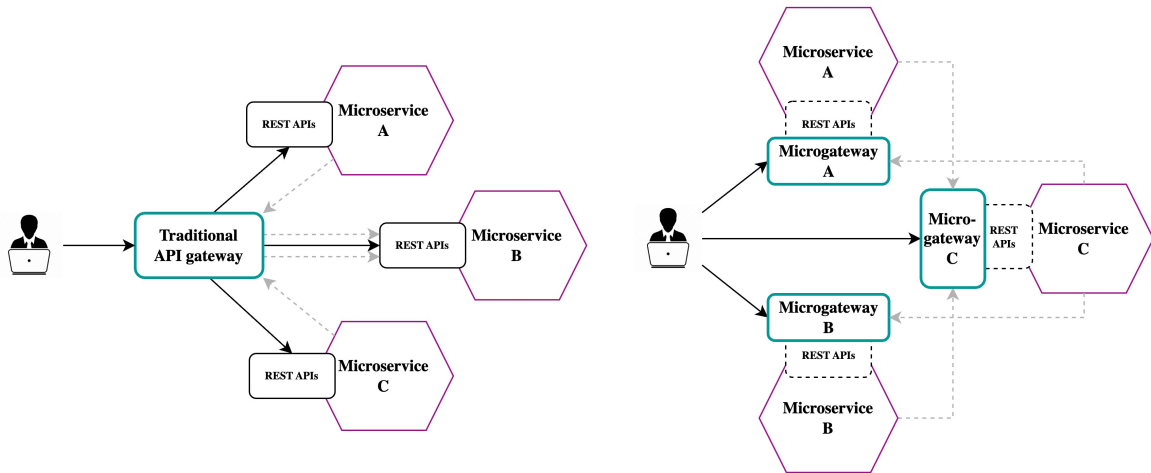


Figure 3.11: Simplified comparison between traditional and monolithic API gateways versus modern and innovative microgateways developed for this thesis.

3.2.2.1 API Microgateways Pattern

The **API Microgateways Pattern** is the infrastructural pattern formalized, further developed and adapted in accordance with the research conducted in this thesis. The primary objective behind the development of this pattern was to design a versatile system of API microgateways capable of being finely tuned to the precise requirements of individual microservices. In the crafting of this pattern, the driving force was the two core principles that underpin its definition and execution:

1. **Customization for microservice needs.** At the heart of the API Microgateways Pattern is the idea of tailoring. Acknowledgment is given to the fact that microservices often have unique and distinct demands. Therefore, the creation of microgateways that

are customized to address the specific needs of each microservice is facilitated by this pattern. This level of customization ensures that the API gateway precisely serves the functionalities required by a given microservice, without any unnecessary or extraneous components. The result is an ecosystem that is finely tuned and highly efficient.

2. **Scalability and adaptability.** The second critical pillar of this pattern revolves around the ability to scale and adapt. As microservices grow and evolve, this pattern is designed to accommodate these changes seamlessly. Scalability is a fundamental consideration, with the pattern being capable of supporting the increasing demands of the microservices landscape. Additionally, it is adaptable, ensuring that it can keep pace with evolving technologies and methodologies. This adaptability is crucial for long-term sustainability in a dynamic and ever-changing technological landscape.

Built upon these two fundamental principles, the API Microgateway Pattern offers a robust and flexible framework for managing and optimizing microservices' interactions with external consumers. It provides the foundation for a responsive, efficient, and customized system that can grow and evolve with the microservices it serves. In the design of the API Microgateway Pattern, as we will see in section 3.3, the choice of a suitable product has been fundamental.

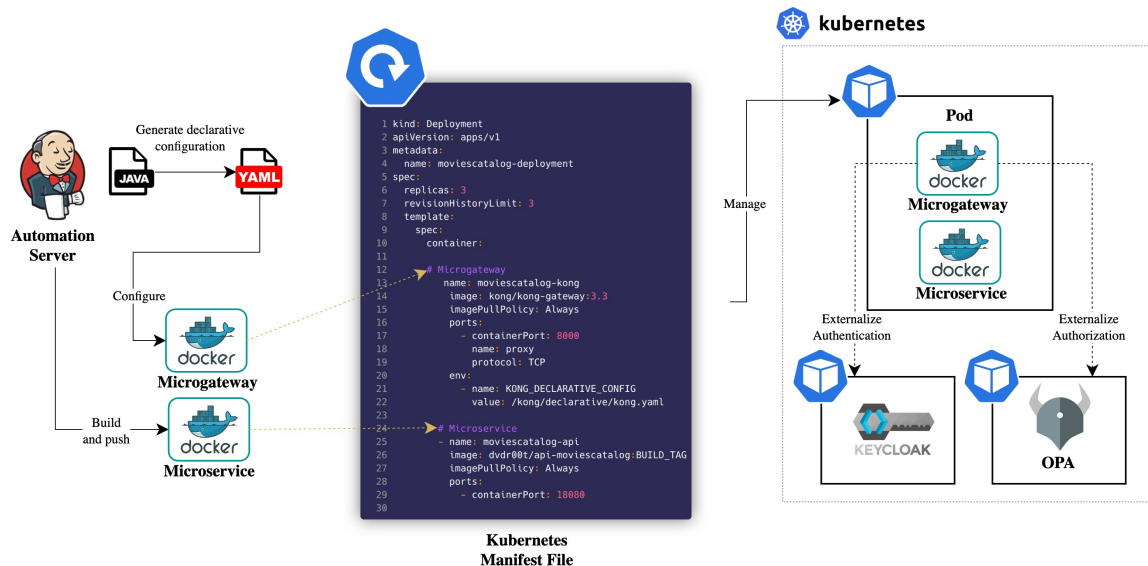


Figure 3.12: Example of configuration of entities in API Microgateway Pattern. The microgateway is lightweight, as it stores the configuration in memory (no DB required). It is rapid in being configured and mounted next to the microservice. And it is flexible as it externalize the supporting businesses to other entities.

Specifically, each microgateway, to accommodate the characteristics previously defined, should have been:

- **Lightweight.** The fundamental principle behind developing a microgateway is to minimize its configuration, enabling it to function as closely as possible to a proxy while delegating supporting functionalities, like authentication and authorization, to external entities. The lightweight microgateway efficiently manages configurations in memory, eliminating the need for a database. This streamlined approach ensures rapid and efficient access to essential information, contributing to faster processing and reduced

latency.

- **Rapid.** Microgateways should exhibit the capability to be swiftly deployed alongside microservices, facilitating rapid and seamless integration without causing downtime. Speed is of paramount importance in today’s dynamic environments, where the ability to adapt quickly is a necessity. Moreover, microgateways should efficiently process network connections, given their interactions with external entities where supporting functionalities are delegated.
- **Flexible.** Microgateways should demonstrate a high degree of adaptability, allowing for easy configuration and customization to meet the specific needs of microservices. The central concept is to employ a declarative format, such as a YAML-based configuration approach, which streamlines the process of making quick adjustments. This flexibility makes microgateways well-suited for seamless integration into continuous integration and continuous delivery (CI/CD) pipelines, enhancing their versatility.

As shown in figure 3.12, adopting a API Microgateway Pattern in CI/CD allows to accomplish all the objectives previously presented and required for a microgateway entity. For instance, the Automation Server (such as Jenkins) can automatically generate declarative configurations for the microgateway based on developers requirements. The configuration can then be loaded in-memory onto the microgateway (what makes it **lightweight**) once it is added to a IaC configuration file, like a Kubernetes manifest. The microgateway will be up and running as soon as the microservice (what makes it **rapid**). Finally, to make the microgateway **flexible**, supporting functionalities such as authentication and authorization can be externalized.

3.2.3 Comparison and considerations

Traditional API gateway	Considerations	API microgateway	Considerations
Monolithic architecture	Unified logic residing in the same entity. It broadly provides all the functionalities of the API Gateway Pattern (authentication, authorization, rate-limiting, etc...).	Micro-architecture	Distributed logic residing in each micro-entity. The functionalities of the API Gateway Pattern are distributed over the whole set of microgateways (each microgateway implements only what the microservice needs).
Centrally managed	Heavyweight entities that are deployed on the infrastructure once and are then able to manage heavy loads of traffic.	De-centrally managed	Lightweight entities that are deployed close to the microservice and are able to manage a restrict load of traffic.
	Scalability issues arise in particular for legacy systems.		Designed with scalability in mind (for instance, to work with orchestration tools such as Kubernetes).
	Single Point of Failure (SPoF).		Distributed concerns over the entire microservices architecture.
Mutable	Mutable object that does not get re-deployed, rather it is re-configured from time to time to adapt it to the APIs needs.	Immutable	Immutable object that does get re-deployed along with the microservices, changing in according to the APIs needs.

Figure 3.13: Table summarizing the differences between the traditional API gateway and the presented API microgateways.

To conclude this section, figure 3.13 summarizes the differences between the traditional API gateway and the presented API microgateways. Traditional API gateways are characterized by a **monolithic architectural** approach where all-encompassing functionalities of the API Gateway Pattern, such as handling authentication, authorization, and rate limiting, are housed within a single entity. They are typically **administered centrally** and are known for their ability to handle substantial traffic loads. However, they can encounter **scalability challenges**, especially when interfacing with legacy systems, and are susceptible to serving as a **single point of failure**. Traditional API gateways are also subject to **periodic reconfiguration** to adapt to changing API requirements.

On the other hand, API microgateways embrace a **micro-architectural design**, with **de-centralized logic** residing in individual micro-entities. Each microgateway is meticulously tailored to address the unique demands of its associated microservice, resulting in a **more lightweight and less centralized** management model. They are engineered with **scalability in mind**, rendering them well-suited for contemporary architectural landscapes and compatible with orchestration tools like Kubernetes. The distribution of responsibilities across the entire microservices architecture mitigates the single point of failure issues, enhancing **fault tolerance**. Moreover, API microgateways **evolve in synchronization with the microservices** they serve, displaying an immutable nature once deployed.

3.3 Adopting Kong as the microgateway technology

In the previous section, we explored the rationale for the **API Microgateway Pattern** and the essential characteristics a microgateway should possess to fulfill its objectives. While a universal and abstract overview of microgateways has been provided, this section will introduce **Kong**, the commercial product adopted for this study. However, it's important to note that Kong is a comprehensive tool with numerous features, and a detailed description of all its aspects is beyond the scope of this research.

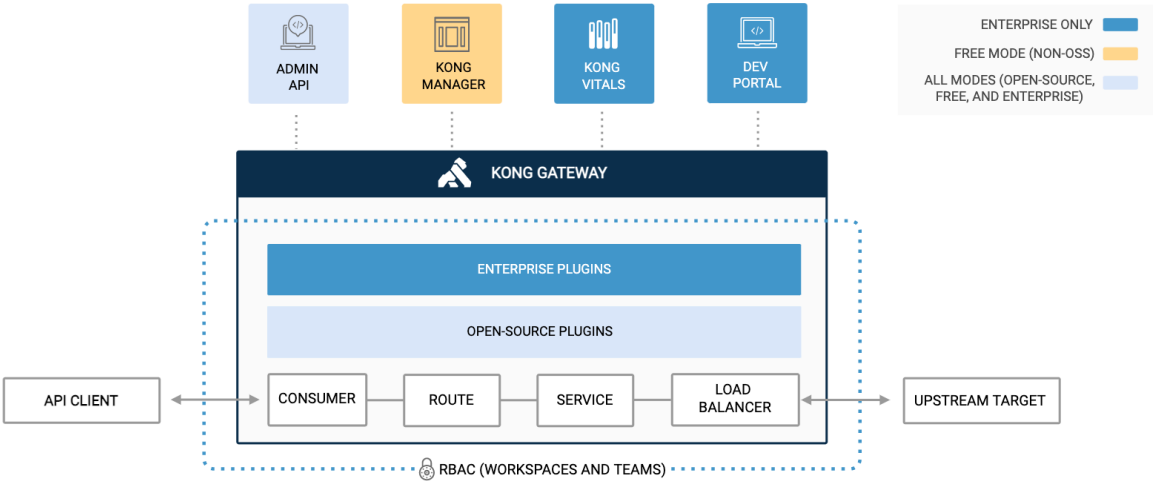


Figure 3.14: Overview of Kong Gateway as traditional API Gateway. Requests originating from an API client enter the Gateway, undergo modification and management by the proxy according to the Gateway’s configuration, and then get forwarded to upstream services. [Source: Kong]

Kong Inc. is a software company that provides open-source platforms and cloud services for managing, monitoring, and scaling APIs and microservices. Some of the products offered include:

- **Kong Gateway.** It serves as a foundational component of the platform, offering robust capabilities for API management and microservices orchestration. Kong Gateway is the product that has been used in this thesis as microgateway.
- **Kong Enterprise.** Built upon the foundation of Kong Gateway, it is an advanced API platform that enhances and extends the capabilities of the gateway. It offers a comprehensive suite of features designed to streamline API management and microservices operations.
- **Kong Konnect.** It is a service connectivity platform that provides a unified approach to connecting and managing APIs and microservices across hybrids and multi-cloud environments. It simplifies the complexity of service communication in a distributed architecture.
- **Insomnia.** It is an open-source tool for API design and testing. This tool, also featured in the thesis, assists developers and teams in designing, documenting, and testing APIs, contributing to the development and validation of robust API solutions.

From 2020, Kong has been named a **Leader** in the **Gartner Magic Quadrant for Full-Lifecycle API Management** ([Pao20], [Sha21] and [Sha22]). Kong’s continued presence as a Leader in these highly regarded reports underscores its enduring excellence and prominence in the field of API management. This recognition is one of the many reasons for the adoption of such product in this thesis.

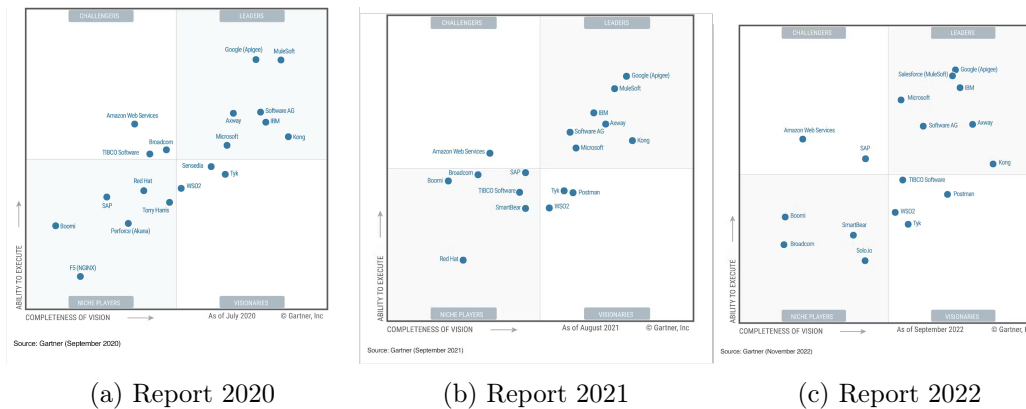


Figure 3.15: Gartner Magic Quadrant(s) for Full-Lifecycle API Management

What’s even more interesting, is the fact that Kong, for three years in a row, has been defined as the most innovative product in the Gartner Magic Quadrants for Full-Lifecycle API Management. Therefore, the study, analysis and adoption of Kong Gateway as a microgateway technology has been a natural consequence of the positive market suggestions. In the upcoming sections, we will try to understand what Kong Gateway is, how it works and why it is different from its competitors in the adoption of this innovative paradigm.

3.3.1 Kong Gateway

Kong Gateway "*is a lightweight, fast, and flexible cloud-native API gateway*" [Kon23h], written in Lua and running in NGINX, that can easily proxy any RESTful microservices exposing APIs. Being written in Lua, it establishes the groundwork for a modular architecture that enables the activation and execution of plugins during runtime. At its core, Kong Gateway incorporates database abstraction, routing, and efficient plugin management. These plugins can reside in separate code bases and can be seamlessly integrated at various stages of the request lifecycle with just a few lines of code.

Generally speaking, Kong Gateway can be deployed either with Kong Konnect or, as it will be present in the rest of this thesis, with self-managed instances. Specifically, the latter modality offers Kong Gateway in two different packages:

- **Kong Gateway Open-Source (OSS)**: an open-source package that encompasses fundamental API gateway functionality along with custom plugins.
- **Kong Gateway Enterprise**: the priced version of the API gateway, with additional functionalities such as Dev Portal, Kong Manager and Enterprise Plugins.

In this thesis, both the open-source and enterprise versions of Kong Gateway have been employed. Chapter 4 (*Enterprise Use-Cases: Securing APIs with Kong*) provides an extensive account of the laboratory environments where both solutions were deployed and tested. It elucidates the rationale behind adopting a combination of on-premise and on-cloud setups to achieve the thesis objectives. However, the following list offers an introductory overview of these two environments without going into the details.

- The initial setup and testing phase has seen the adoption of Kong Gateway Open-Source (OSS). A **microk8s** (Kubernetes) cluster was established within the servers of **Liquid Reply** (IT) company. Kong Gateway was meticulously examined to create a robust framework intended for use as a microgateway.
- Given the open-source version's limitations, particularly in terms of available plugins, the Enterprise edition was adopted. To further design and simulate real-world scenarios, an **Azure Kubernetes Service** (AKS) cluster was set up within an Azure Subscription provided by **Spike Reply** (DE).

Kong Gateway perfectly fits in the API Microgateway Pattern, thanks to its **lightweight, fast and flexible** nature. Specifically, as we will see later in this section, Kong Gateway offers the possibility to store gateway configurations in memory, without the need of a DB. This feature allows rapid deployment of immutable microgateway that does not need synchronization with an external database. Moreover, by adopting a declarative format to store the configurations, Kong Gateway is best suitable to work in CI/CD scenarios, where APIs are automatically tested and transformed in Kong files.

3.3.1.1 How it works

Kong Gateway provides, at its core, proxying capabilities by exposing several interfaces which can be tweaked in accordance to the microservices needs. Specifically:

- `proxy_listen` defines a list of addresses (hosts and ports) on which Kong Gateway accepts public HTTP traffic from clients and proxy it to the upstream services.

- `admin_listen` defines a list of addresses (hosts and ports) on which Kong Gateway accepts administrator requests, exposing over HTTP the set of admin APIs.

The following terms, in accordance with the Kong Gateway terminology [Kon23k], will also be useful in the upcoming chapters:

- A `client` refers to the *downstream* client, i.e., the **entity making the requests** to the Kong microgateway proxy port. To test every proof of concept of this thesis, as it will be presented in Chapter 4, multiple clients have been used: from simple cURL to Insomnia and Postman, but also Web Browsers, Third-Party applications and ad-hoc developed applications.
- An `upstream_service` refers to the **application running in the microservice** exposed by Kong microgateway, to which `client`s are eventually redirected by Kong. In this thesis, a MoviesCatalog Spring Boot CRUD application (4.1) serves as the `upstream_service` for most of the use-cases.
- A `Service` is a Kong **entity that abstracts the `upstream_service`** within the Kong microgateway's configuration.
- A `Route` is a Kong **entity that defines the exposed entry-points** of Kong microgateway. It defines the rules that regulates requests matching and forwarding to a given `Service`.
- A `Plugin` refers to **Kong Gateway plugins**, i.e. customized logic that elaborates the incoming requests. Plugins will be configured in a declarative fashion, both globally (all incoming traffic) and on specific routes and services.
- A `Consumer` is a Kong **entity that may be either anonymous or associated with a specific client**. Kong microgateway's consumers may define specific credentials in order to be identified. As we will see in Chapter 4, by not having a database to persistently store consumers, specific usages of this entity have been adopted.

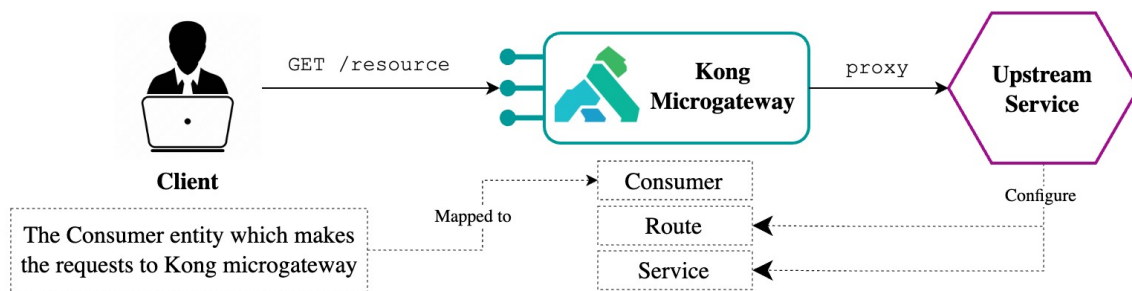


Figure 3.16: Overview of Kong microgateway terminology. Clients, which may be associated to a consumer, make requests based on the configured routes. The request is mapped to the specific service and proxied to the correspondent upstream.

From a high-level perspective, Kong microgateway actively monitors HTTP traffic on its pre-defined proxy ports, which are set to `8000` and `8443` by default. In this role, Kong microgateway assesses each incoming HTTP request against the configured routes to identify a suitable match. Once a request aligns with the criteria of a specific route, Kong Gateway proceeds

to proxy the request. Each route may be associated with a service, and in such cases, Kong Gateway proceeds to execute the plugins configured for both the route and the corresponding service. After plugin execution, it then proxies the request upstream.

3.3.1.2 Admin APIs

Kong Gateway incorporates **internal REST APIs** [Kon23a] that can be used to configure and obtain information about the microgateway. Specifically intended for administrative tasks, Admin APIs listen by default on port 8001, while port 8444 is reserved for secure HTTPS traffic. It's essential to emphasize that those APIs are tailored for internal use and grant comprehensive control over Kong. Consequently, when configuring Kong microgateways, it is crucial to exercise caution to prevent any unintended public exposure of these APIs.

However, one of the consequences of adopting a DB-less Kong Gateway as microgateway, is that Admin APIs are considerably restricted. Specifically, in DB-less mode, Admin APIs serves two primary functions. First, they facilitate the loading of a new declarative configuration, and secondly, they enable inspection of the current configuration. In this mode, the Admin API operates independently for each individual Kong microgateway, mirroring the memory state specific to that particular microgateway. **In DB-less mode, Kong Gateway is configured declaratively.**

Admin APIs have been used in this thesis to perform the following operations:

- **Load new configuration.** Kong Gateway exposes the `POST /config` endpoint to load a new declarative configuration inside the memory. By providing a declarative YAML file (in this thesis, called `kong.yaml`), Kong re-configure the set of Services, Routes, Plugins, Consumers and the other entities declared.
- **Retrieve entities.** Kong Gateway exposes Kong entities information to the correspondent specific endpoints (`GET /services/:serviceName`, `GET /routes/:routeName` or similar for the other entities). In DB-less mode, **Admin APIs are mostly read-only.** Administrators and developers can interact with the Admin APIs to retrieve information about the status of the microgateway's configuration.
- **Validate a configuration.** Kong Gateway exposes a specific endpoint to validate the defined configurations. At `POST /schemas/{entity}/validate` it is possible to validate entities against the Kong pre-defined schema. Although not used directly in this thesis, this is one of the endpoints used by **deck** to validate the `kong.yaml` generated configuration.

3.3.1.3 Declarative configurations

As stated earlier, using both the DB-less mode and declarative configuration offers several advantages for adopting Kong Gateway as a microgateway technology:

- **Reduced number of dependencies:** in DB-less mode there is no need to manage a database installation. This is best suitable in a microgateway scenario, in which microgateways should be autonomous enough to be disconnected and re-plugged as needed, without managing database credentials and access controls.
- **Automation in CI/CD:** configuring entities can be centralized in a single source of truth, managed through a Git repository. In addition to that, for this study, the configu-

rations are automatically generated to improve speed and reliability in the development of new APIs. By validating and loading the configuration file at deployment time in CD, Kong Gateway fits well in this paradigm.

- Finally, as stated in [Kon23b], by not being dependent on specific database technologies, it enables a wider range of use-cases in which Kong Gateway can be adopted as microgateway.

The fundamental idea behind declarative configuration is its non-imperative nature. Unlike imperative configurations, which are given as a set of step-by-step instructions ("do this, then do that"), **declarative configurations** are presented all at once, declaring the desired end state of a system. A declarative configuration file encapsulates settings for all target entities within a single document. Upon loading this file into Kong microgateway, it entirely replaces the existing configuration. When incremental changes are needed, they are applied to the declarative configuration file, which is subsequently reloaded as a whole in CI/CD. This ensures that the system's configured state always aligns with the description in the loaded file which is kept under Version Control in a Git repository.

The following piece of code is an example of declarative configuration defined manually for the initial steps of this thesis. In section 4.1.2.3 we will see how to automatically generate Kong configuration files starting from an OpenAPI Specification file.

```
1  _format_version: "3.0" # specifies the number of the declarative config
2  _transform: true # encrypt or hash config before importing credentials
3
4  services: # defines the upstream service entity
5  - name: moviescatalog-service
6    protocol: http
7    host: localhost
8    port: 18080
9    plugins:
10   - name: key-auth # these plugins apply only to this service
11  routes: # defines the routes for the service entities
12  - name: retrieve-all-movies
13    paths:
14    - /movies
15
16  consumers:
17  - username: keycloak
18    keyauth_credentials:
19    - key: my-key
20
21  plugins: # these plugins apply globally
22  - name: prometheus
23    enabled: true
24    config:
25    per_consumer: false
26    status_code_metrics: true
```

As seen in the presented declarative configuration file, Kong plugins have their own section.

The next section delves into the details of that.

3.3.2 Kong Plugin Hub

To extend Kong microgateway with additional features, a set of specific plugins can be applied to the configuration file. While official plugins can be found in the [Kong Plugin Hub](#), Kong Gateway allows the definition of custom plugins written in Lua, Go or even Python.

Plugins provide Kong microgateway with the **flexibility to adapt to various microservices**. For example, in this thesis, we've developed enterprise scenarios where Kong microgateway had to possess the capability to offer data for Prometheus scraping, establish authentication and authorization flows, manage rate-limiting for requests, and more. However, since each microgateway serves a distinct microservice, each of which typically has unique requirements and functionalities, plugins enable Kong microgateway to be tailored to specific needs.

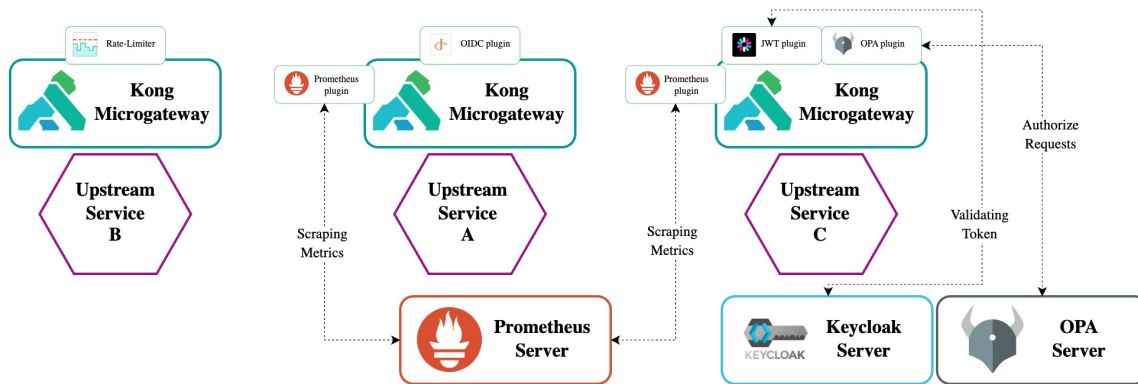


Figure 3.17: Three different example of Kong microgateway adapted to three different microservices. Kong Gateway instance remains the same, but the declarative configuration defined in CI/CD loads different plugins for each use case.

For example, let's take a look at figure 3.17, which illustrates three distinct scenarios. What's essential to observe is that the Kong microgateway remains consistent at its core for all upstream services. What differs are the plugins that are configured in a CI/CD pipeline to manage it. The ability to attach these plugins to Kong in a declarative manner is a key factor driving the adoption of Kong Gateway as a microgateway technology in this thesis. **Each microgateway is tailored to the specific requirements of the microservice it serves.**

To illustrate, only the microservices needing metrics scraping will have Prometheus plugins added to the Kong microgateway. These Prometheus servers exist externally to Kong for centralized configuration. A similar approach is taken for Keycloak and the Open Policy Agent (OPA). Only those microservices requiring authentication and authorization capabilities are enhanced with the JWT and OPA plugins. This feature allows Kong to be a **lightweight** microgateway that delegates supporting businesses to external entities.

Specifically, the following list comprehends all the plugins used during the course of this thesis:

- **Authentication Plugins:** the set of plugins adopted to manage the authentication-related aspects for the consumers of the APIs. Different scenarios based on these plugins

are analyzed in section 4.4.

- **Basic Authentication**. Basic Authentication plugin allows to add **basic authentication** for requests based on consumer’s username and password. While it’s a straightforward use case and not considered an industry-standard practice, it effectively demonstrates Kong microgateway’s configuration capabilities.
- **Key Auth**. The API Key Authentication plugin allows for the integration of **API key-based authentication** into a service or route. Consumers can authenticate their requests by including their API key in a query string parameter, a header, or within the request body. Key Auth is one of the simplest form of authentication, but still widely adopted, especially in machine-to-machine communication where no user is involved.
- **JWT**. The JWT plugin provides the capability to validate requests containing **JSON Web Tokens** (JWTs) signed with either HS256 or RS256, adhering to RFC 7519 standards. When activated, the plugin assigns JWT credentials, including public and secret keys, to consumers, which they use to sign their JWTs. These tokens can be transmitted via various methods, such as query string parameters, cookies, or HTTP request headers. Kong’s role is to either route the request to upstream services upon successful token signature verification or reject it if verification fails. Additionally, Kong can perform verifications on specific registered claims from RFC 7519, such as `exp` and `nbf`.
Within this thesis, this is unequivocally the most commonly adopted plugin for validating all Proof of Concept (PoC) infrastructures. This plugin serves as a straightforward yet highly adaptable tool, enabling external entities like Keycloak, acting as an Authorization Server, to generate JWTs that will subsequently undergo validation by Kong.
- **OAuth 2.0 Authentication**. The OAuth 2.0 Authentication plugin transforms Kong microgateway in an OAuth Authorization Server, enabling the classic OAuth 2.0 Authorization flows (Client Credentials, Authorization Code, Implicit and Resource Owner Password). However, because the core objective in an API Microgateway Pattern is to externalize the logic from the gateway to external entities, this plugin will be presented simply for completeness reasons.
- **OpenID Connect** [*Enterprise only*]. The OpenID Connect (OIDC) plugin enriches Kong with OIDC capabilities, overseeing not only Access Tokens and Refresh Tokens but also the management of **ID Tokens**. It serves as a seamless expansion of the JWT plugin and can be employed as an alternative, enabling the fulfillment of OIDC functionalities.
- **SAML** [*Enterprise only*]. The SAML plugin in Kong serves a critical role within the **Security Assertion Markup Language** (SAML) framework, an open standard that simplifies the interchange of authentication and authorization data between different identity providers (IdP) and a service provider (SP). Within the SAML specification, there are three fundamental roles: the principal, the identity provider (IdP), and the service provider (SP). In this context, the Kong SAML plugin functions as the service provider, primarily tasked with initiating a login to the identity provider, which is referred to as an **SP-Initiated Login**, thereby playing a pivotal part in the secure authentication and authorization flow between entities.

- **Security Plugins:** the set of plugins selected for managing authorization-related aspects for the consumers of the APIs. Different scenarios based on these plugins are analyzed in section 4.4.
 - **OPA** [*Enterprise only*]. The OPA (**Open Policy Agent**) plugin in Kong is a vital component that forwards incoming requests to the OPA servers for evaluation. This evaluation step is conditional; it proceeds only if the specified authorization policy permits the request. In essence, the OPA plugin plays a crucial role in augmenting Kong’s capabilities by **offloading complex authorization logic** to the Open Policy Agent, ensuring that requests are meticulously processed according to defined policies, thereby enhancing security.
- **Traffic Control Plugins:** the set of plugins selected for enabling traffic control and restriction functionality in Kong microgateways. An implementation scenario based on these plugins is analyzed in section 4.3.
 - **Rate Limit.** The rate-limiting plugin in Kong provides a pivotal functionality by governing the number of HTTP requests that can be made within specified time intervals, whether in seconds, minutes, hours, days, months, or even years. This mechanism is adaptable to the presence of an authentication layer within the underlying service or route. In cases where no authentication is in place, the Client IP address is employed for rate limiting. However, if an authentication plugin has been configured, the consumer’s credentials are utilized to enforce rate limits, ensuring a fine-tuned control over request limits based on the authentication context.
- **Analytics & Monitoring:** the set of plugins selected for enabling observability and monitoring functionality in Kong microgateways. An implementation scenario based on these plugins is analyzed in section 4.3.
 - **Prometheus.** The Kong Prometheus plugin exposes Kong and upstream service **metrics in Prometheus format**, suitable for scraping by a Prometheus Server. These metrics are accessible via the `http://localhost:{port}/metrics` endpoint on both the Admin and Status APIs. To utilize this functionality, the Prometheus server should employ a service discovery mechanism to collect data from each Kong node’s `/metrics` endpoint, allowing for comprehensive monitoring.

In conclusion, it’s important to note that the list of plugins presented here represents only a fraction of the available options within Kong’s extensive ecosystem. The Kong Plugin Hub hosts a wealth of additional plugins, each catering to specific needs and use cases. Furthermore, the potential for custom plugins adds another layer of innovation to Kong’s capabilities, allowing for the exploration of novel features and integrations that can further enhance Kong’s utility. As such, future endeavors and research may involve evaluating these custom plugins to unlock even more potential in the realm of API management and beyond.

3.3.3 Microgateway approach in DevSecOps

To conclude this chapter, the adoption of Kong Gateway as microgateway technology is going to be discussed. In the previous section, the **flexible** capabilities were elaborated upon through the utilization of plugins, and the **lightweight** nature was demonstrated through the implementation of in-memory configuration in DB-less mode. In this section, the **rapid**

functionality of Kong microgateway, which can be easily employed in CI/CD scenarios, will be explained.

To achieve this objective, among the multitude of technologies adopted, we will leverage two key components:

- `deck` automation tool (from Kong).
- Kubernetes integration to load declarative configurations.

3.3.3.1 `deck` automation tool

`deck`, standing for *declarative Kong*, streamlines Kong configuration management through a declarative approach, empowering developers to define their desired states, encompassing services, routes, plugins, and more. With `deck`, the **implementation process is automated** [Kon23c], sparing the manual execution of each step, as is common with the Kong Admin API. It boasts several notable features, such as configuration synchronization with running Kong clusters, drift detection for manual changes, configuration backups, the establishment of automation pipelines with APIOps, and decentralized configuration management using tags, enabling efficient collaboration among diverse teams for configuration distribution. Within this thesis, `deck` has been adopted to enable APIOps [Kon23d] in the early stage of the design phase.

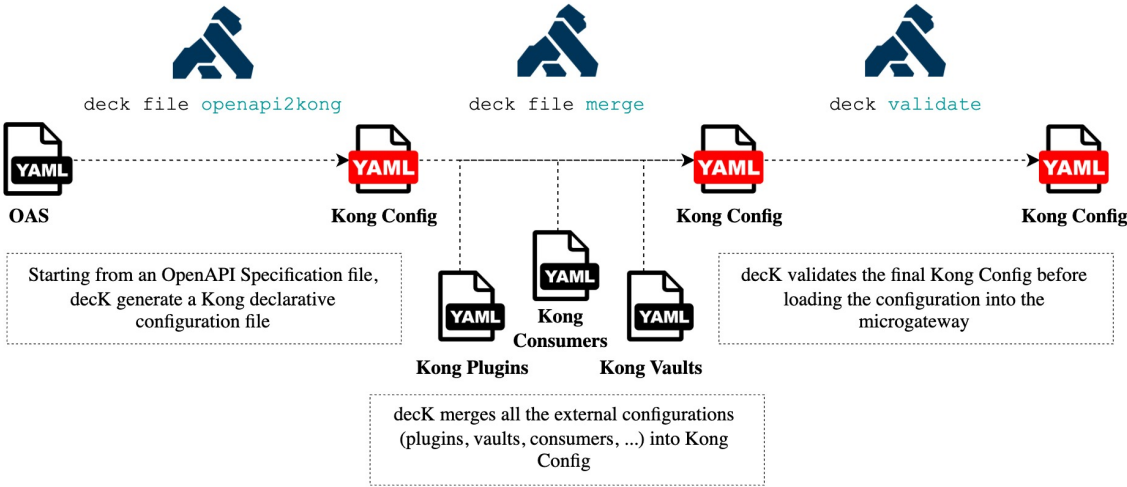


Figure 3.18: Adapting `deck` to the **API Microgateway Pattern**. Developers can now keep microgateway configurations under Version Control in the microservice-dedicated Git repository. Each configuration is tailored to the microservice and `deck` manages the integration and validation of the Kong declarative Config.

In figure 3.18, the depicted sequence outlines the steps that we have adopted to involve `deck` in the automated configuration management of Kong microgateways. This process unfolds within the realm of CI/CD, where each microservice resides within its own dedicated Git repository. These repositories host essential elements, including the OpenAPI Specification file (as will be elaborated upon in Section 4.1.2.3), and custom entities designed for the specific Kong microgateway, such as plugins, consumers, and Vault integration. The following steps delineate the process:

1. At first, **decK converts the OpenAPI Specification file into a Kong microgateway configuration** [Kon23e]. OpenAPI [Sma] is the most commonly used standard for defining API behavior, and the adoption of decK in the configuration of each Kong microgateway allows to reach industry-level standards.

```
1 deck file openapi2kong \  
2   --spec build/openapi.yaml \  
3   --format yaml \  
4   --output-file kong.yaml
```

The result of such operation would be a `kong.yaml` file similar to the one presented in 3.3.1.3. However, in this case, Kong entities like consumers, plugins, vaults and more, cannot be extracted from the OAS file and therefore will not be present. The next step solves this problem.

2. In an API Microgateway Pattern context, where every microgateway is customized to the unique requirements of each microservice, the utilization of decK is a strategic choice to consolidate all configurations into a single consolidated file [Kon23f]. Through the application of decK's file transformation commands, it becomes possible to structure Kong configuration files into **distinct segments of the complete configuration**, subsequently merging them before loading into the microgateway itself. This approach enables the systematic organization of various configuration elements in accordance with the developers team. Specifically, starting from the `kong.yaml` previously generated:

```
1 deck file merge kong.yaml \  
2   kong-plugins.yaml \  
3   kong-vaults.yaml \  
4   kong-consumers.yaml \  
5   --output-file kong.yaml
```

The resultant file constitutes the ultimate configuration, primed for seamless integration with the Kong microgateway. However, validation is required before proceeding to the deployment.

3. To verify the configuration, decK once again assumes a pivotal role. The **validate command operates by analyzing the state file, confirming its correctness**. It meticulously scans all designated state files, highlighting any YAML/JSON parsing discrepancies, and systematically assesses foreign relationships, signaling potential issues such as broken connections or missing links.

```
1 deck validate --state kong.yaml
```

decK stands as a continuously evolving and power tool, playing a pivotal role in this thesis. Its adoption has been instrumental in achieving the three fundamental objectives outlined for the API Microgateway Pattern: **lightweightness**, **speed**, and **flexibility**. decK facilitates the management of external configurations in a highly adaptable manner, effectively segregating concerns and automating otherwise laborious tasks. Furthermore, it offers the capability to rapidly transform configuration files, which are subsequently stored in Kong's memory for efficient execution.

3.3.3.2 Integration with Kubernetes

In this concluding section, the integration between Kong microgateway and Kubernetes is presented. As elaborated in section 3.1.3, Kubernetes serves as a pivotal container orchestration tool employed in this thesis to effectively oversee a swarm of microgateways and microservices. At the heart of the API Microgateway Pattern lies the principle that microgateways should maintain proximity to their respective microservices. As seen in figure 3.8, our chosen approach revolves around Kubernetes deployments, orchestrating a cluster of Kubernetes **Pods that contain both the microgateway and the microservice**.

Because everything is managed in CI/CD pipelines, the desired state of the deployment is configured in a `kubernetes-deployment.yaml` file, adopting principles from Infrastructure as Code (IaC) best-practices. The configuration file is kept in the Git repository under Version Control, so that developers can collaboratively manage it. The Kubernetes deployment is **rolled-out** at each build, installing in the cluster both the new version of the microservice exposing the APIs and the newly configured microgateway. Specifically, we'll see how to **load declarative configuration** at start-up exploiting Kubernetes `ConfigMap`.

Building upon the previous step in the CI/CD workflow, the agent running the pipeline should possess a `kong.yaml` file encompassing the configuration specifics for the Kong microgateway. When re-deploying the microgateway, Kong's operation involves retrieving the updated configuration file from the Kubernetes cluster. At first, the `ConfigMap` entity will be updated (see 5): it will now contain the new configuration for Kong. The Deployment entity in the Kubernetes file contains the following `Volume`:

```
1 volumes:
2 - name: kong-config-volume-moviescatalog
3   configMap:
4     name: kong-config-moviescatalog
```

The `Volume` is mounted into the container's filesystem (at `/kong/declarative`) running Kong microgateway as:

```
1 volumeMounts:
2 - name: kong-config-volume-moviescatalog
3   mountPath: /kong/declarative
4   readOnly: true
```

Finally, Kong Gateway allows the definition of an environment variable, `[Kon23g]` called `KONG_DECLARATIVE_CONFIG`, to load the declarative configuration file at Kong start-up. The container definition becomes:

```
1 env:
2 [...]
3 - name: KONG_DECLARATIVE_CONFIG
4   value: /kong/declarative/kong.yaml
```

The provided code snippet offers insights into the fundamental and core elements within the `kubernetes-deployment.yaml` file. To enhance clarity and comprehensibility, comments have been included.

```

1 kind: Deployment
2 apiVersion: apps/v1
3 [...]
4 spec:
5   [...]
6   template:
7     spec:
8       [...]
9       volumes:
10        - name: kong-config-volume-moviescatalog
11          configMap:
12            name: kong-config-moviescatalog
13          [...]
14        containers:
15
16          # Microservice exposing the APIs (microservice-container)
17          - name: moviescatalog-api
18            image: dvdr00t/api-moviescatalog:BUILD_TAG
19            imagePullPolicy: Always
20            ports:
21              - containerPort: 18080
22
23          # Kong microgateway (microgateway-container)
24          - name: moviescatalog-kong
25            image: kong/kong-gateway:3.3
26            imagePullPolicy: Always
27            ports:
28              - containerPort: 8000
29                name: proxy # proxy port for consumers incoming requests
30                protocol: TCP
31              [...]
32              - containerPort: 8100
33                name: metrics # port for scraping by prometheus
34                protocol: TCP
35            env:
36              [...]
37              - name: KONG_DATABASE
38                value: "off"
39              [...]
40              - name: KONG_DECLARATIVE_CONFIG
41                value: /kong/declarative/kong.yaml
42            volumeMounts:
43              - name: kong-config-volume-moviescatalog
44                mountPath: /kong/declarative
45                readOnly: true
46            [...]

```

4. Enterprise Use-Cases: Exposing APIs with Kong microgateway

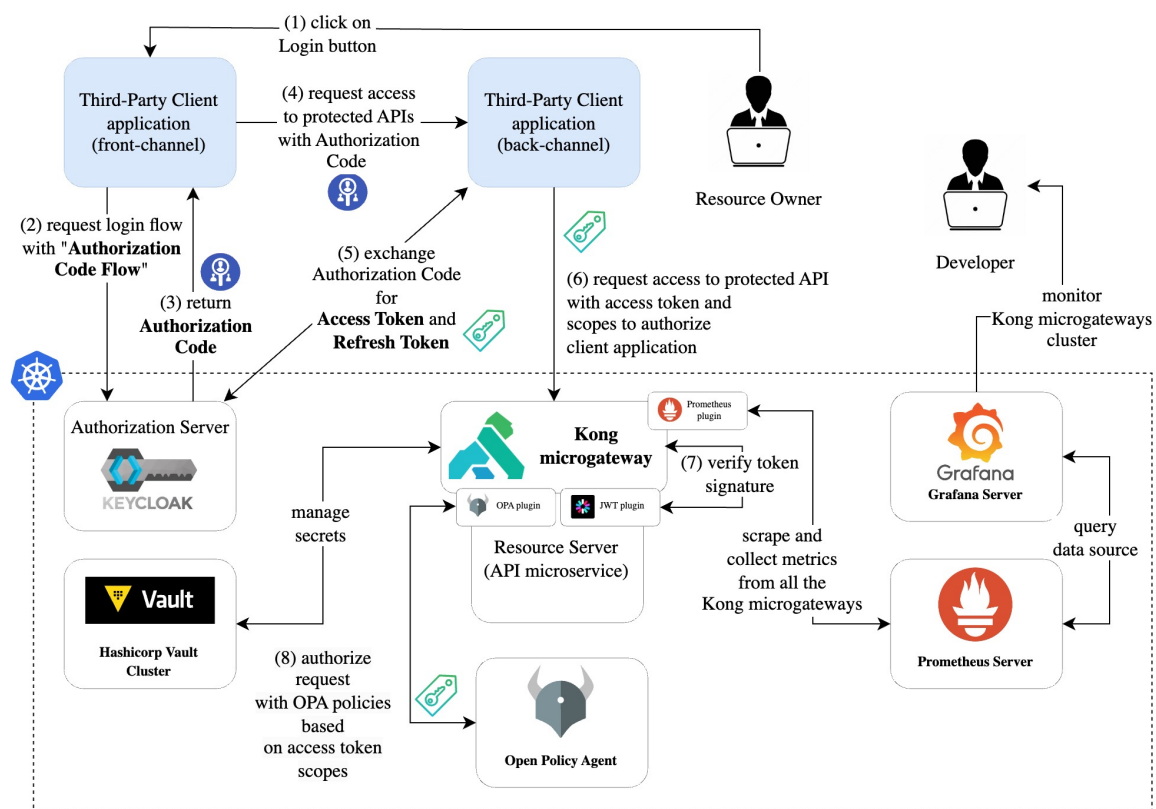


Figure 4.1: Example of a complete enterprise-ready scenario. Kong microgateway is integrated within an OAuth 2.0 Authorization Code Flow adopting Keycloak as the Authorization Server and OPA as the policy enforcement framework. HashiCorp Vault integration is adopted to manage confidential secrets, while Prometheus and Grafana monitors traffic flow in the cluster.

Chapter 2 and Chapter 3 have collectively established the essential background framework for this thesis. The former has provided the necessary context by introducing the key concepts and tools related to the APIs lifecycle, such as the OpenAPI Specification format and the DevSecOps best-practices adopted in the APIOps paradigm. It set the stage for the latter chapter, which introduced the innovative **API Microgateway Pattern**, along with the core pillars that microgateways should possess. Furthermore, it illustrated the integration of

Kong Gateway within this paradigm, analyzing the product’s essential features that lead the decision for its adoption.

This chapter presents the technical details of the **enterprise use-cases** that have been analyzed, implemented, and rigorously tested throughout this study. These practical scenarios shed light on the real-world application of the API Microgateway paradigm within the dynamic landscape of modern enterprises. Kong microgateway will be showcased in several and different scenarios. Specifically, this chapter will focus on Kong microgateway integration with Authentication and Authorization industry standards, such as OAuth 2.0, OpenID Connect and SAML 2.0. Not limited to that, scenarios for secrets management, traffic control, monitoring and observability will be presented.

Although figure 4.1 portrays a potential, enterprise-ready, and comprehensive scenario, this chapter breaks down individual component features to isolate distinct concepts and enhance readability. Chapter 6 will assess a similar scenario, highlighting not only the infrastructure responsible for APIs exposure but also the integration of automation tools within the scenario.

Before delving into the details of the mentioned scenarios, an introduction to the microservice that was developed and used to test the scenarios is provided.

4.1 MoviesCatalog: a simple CRUD microservice

MoviesCatalog serves as a basic and straightforward **Gradle-based Spring Boot Web Application**, developed in Kotlin to offer **CRUD functionalities** for efficiently managing a collection of movies. This application empowers users to access, create, update, and remove movie entries. As it is designed only for testing purposes, it does not offer a database connection, opting for loading and storing movies into the in-memory H2 database. However, the reasons behind the development of such application rather than adopting out-of-the-box solutions stands by the complete control over the exposed APIs. This enables the usage of the key tools adopted in the development of this thesis, including Gradle, 42Crunch, and decK.

4.1.1 The application

The application is designed and developed to run within **Docker containers** and be deployed on a **Kubernetes cluster** using Kong microgateway to expose the APIs. In alignment with the microservices paradigm (3.1), it focuses solely on core functionalities, outsourcing supporting functions to external entities: authentication is overseen by Keycloak, policy enforcement is handled by the Open Policy Agent, monitoring is orchestrated through Prometheus, and secrets storage is managed either in CI/CD pipelines by Jenkins or GitHub Actions, or integrated with HashiCorp Vault through the Kong microgateway.

Specifically, the application adopt the **Controller-Service-Repository pattern**, in which the `MoviesController` exposes the endpoints and manages the REST interface to the business logic, which is implemented by the `MoviesService`. Finally, the `MoviesRepository` represents the persistence layer and interacts with the JPA APIs.

4.1.1.1 Core plugins

MoviesCatalog adopts Gradle as the central management tool throughout its entire lifecycle. As we explore in the upcoming sections (4.1.2), the following plugins play a vital role in seamlessly integrating its functionalities within the **API Microgateway paradigm**:

- **JUnit 5 plugin.** The JUnit testing framework plugin facilitates the creation and execution of Java tests, ensuring efficient and reliable testing for Java applications. Although the integration of this plugin does not serve the API Microgateway Pattern objective, as shown in 4.1.2.1, it completes the process of developing APIs with industry-level standards.
- **Google Jib plugin.** The `id("com.google.cloud.tools.jib")` version "3.1.4" plugin builds optimized Docker images for Java applications, offering fast, reproducible, and daemonless container image generation. As shown in 4.1.2.2, the adoption of Jib simplified the streamline processes to integrate pushed source code into new Docker images ready for deployment.
- **OpenAPI plugin.** The `id("org.springdoc.openapi-gradle-plugin")` version "1.6.0" plugin allows to generate an OpenAPI 3 Specification file for each Gradle build, adapted to the Spring Boot application. As shown in 4.1.2.3, this plugin automatically generate the OAS file that will be validated with 42Crunch Security Audits and converted to Kong configurations.
- **Gatling plugin.** The `id("io.gatling.gradle")` version "3.9.5" plugin enables load testing for web applications, allowing for robust performance testing and analysis. As shown in 5.3.2.3, scenarios simulations in Gatling allowed to asser the performances of the infrastructure.

Moreover, as already mentioned, as the application is developed only for testing purposes, in order to facilitate deployment and focus on the microgateway functionalities, it is not connected to any external database. *Persistence* is provided only in-memory, therefore working as long as the Kubernetes Deployment entity hosting the application does to get rolled-out. Specifically, data storage for the presented scenarios happens through a [Spring Boot H2 Database](#) that is loaded at start-up with a set of pre-defined values.

```

1 spring.datasource.url=jdbc:h2:mem:testdb
2 spring.datasource.driverClassName=org.h2.Driver
3 spring.datasource.username=sa
4 spring.datasource.password=
5 spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

```

The upcoming sections will showcase the usage of the listed tools.

4.1.1.2 APIs

MoviesCatalog application implements the following REST APIs.

- `GET /movies` allows to fetch movies, providing filter options based on the movies genre and responding with a pagination mechanism.
- `POST /movies` allows to add a new movie to the catalog. As presented in Chapter 7 (*Conclusions and Future Work*), entity validation can be also implemented at the microgateway level by the adoption of the [Request Validator](#) plugin for Kong microgateway.
- `PATCH /movies/:name` allows to update the content of an existing movie. As for the `POST`, it expects an HTTP JSON body containing the new version of the movie.

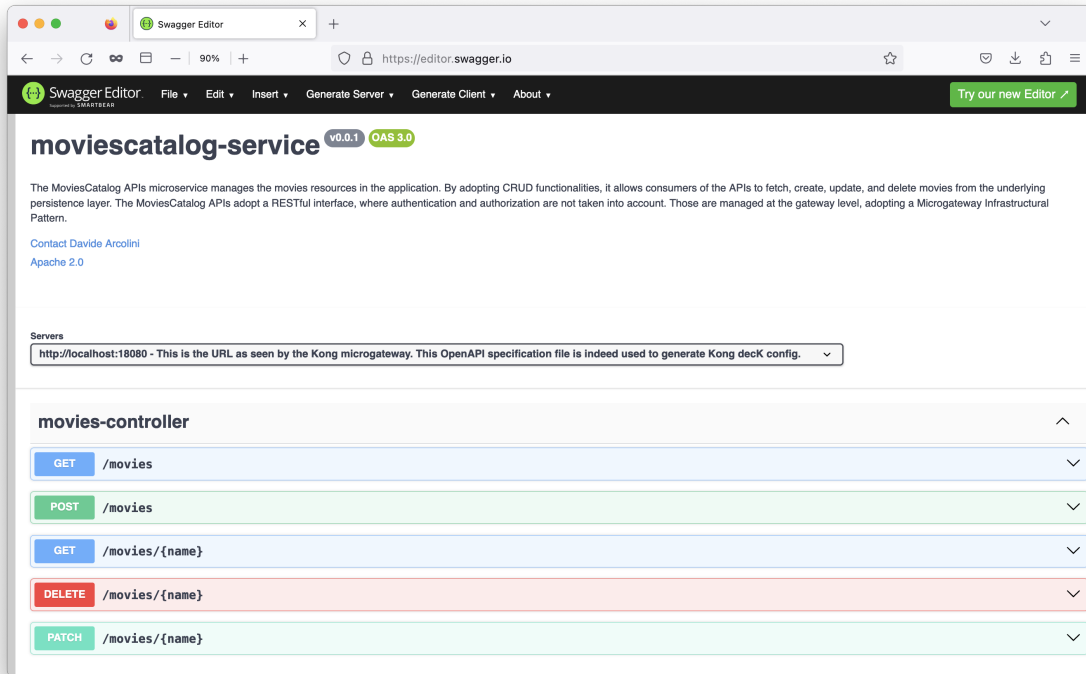


Figure 4.2: OpenAPI Specification overview of the *MoviesCatalog* application.

- **DELETE** `/movies/:name` allows to remove a movie from the catalog.

In the presented scenarios, the **POST**, **PATCH** and **DELETE** operations will required API consumers to be authenticated and authorized in order to access the catalog. This functionality, in accordance with the **API Microgateway Pattern** is managed at the microgateway level by Kong. Moreover, all the APIs adopt the HTTP response status codes defined by RFC 9110 [FNR22], allowing for a clear and efficient definition of the Grafana dashboard visualization during monitoring scenarios.

4.1.2 Tasks automation with Gradle

As shown in section 4.1.1.1, the adoption of Gradle as management tool allows the definition of various plugins that improve the integration of this microservice with Kong microgateway. Specifically, this section presents:

- The **JUnit 5** testing framework integration.
- The automatically generation of the **OpenAPI Specification** files.
- The continuous integration of Docker images into the remote registry with **Google Jib**.

4.1.2.1 JUnit 5 tests

JUnit [Ken23] is a testing framework that facilitates the creation and execution of Java tests, ensuring efficient and reliable testing for Java applications. Although this is clearly not the focus of this research, two test-suites have been provided for testing purposes, in order to consolidate the concepts of the Full-Lifecycle API Management in which unit and integration tests should be run as soon as possible in accordance with the DevSecOps paradigm. The two

test-suites are the following:

- `MoviesControllerTests` tests the implementation of the `MoviesController`. It specifically asserts that each API returns the expected response.

```
1  @Test
2  fun `Retrieve all movies`() {
3      val expectedResult: MutableList<MovieDTO> =
4          ↪ mutableListOf<MovieDTO>()
5      repeat(5) {
6          expectedResult.add(
7              moviesRepository.save(moviesFactory.generateMovieModel())
8                  .toDTO()
9          )
10     }
11     val response: ResponseEntity<String> = restTemplate.exchange(
12         "/movies",
13         HttpMethod.GET,
14         HttpEntity(null, null),
15         String::class.java
16     )
17     val objectMapper = jacksonObjectMapper()
18     val responseBody: PageResponseDTO<MovieDTO> =
19         ↪ objectMapper.readValue(response.body, object:
20         ↪ TypeReference<PageResponseDTO<MovieDTO>>() {})
21
22     /* evaluating assertions */
23     Assertions.assertNotNull(response)
24     Assertions.assertEquals(HttpStatus.OK, response.statusCode)
25     Assertions.assertEquals(5, responseBody.content.size)
26     Assertions.assertEquals(expectedResult.toList(),
27         ↪ responseBody.content)
28 }
```

- `MoviesServiceTests` tests the implementation of the `MoviesService`. It specifically asserts that the business logic of the application does not change when the APIs implementation is changed and new code is pushed to the remote repository.

```
1  @Test
2  fun `Fail to retrieve a non existing movie`() {
3      /* assertions evaluation */
4      val exception = assertThrows<MoviesExceptions.MovieNotFound> {
5          moviesService.retrieveMovieByName("nonExistingName")
6      }
7      Assertions.assertEquals("No matching movie with the given name.",
8          ↪ exception.message)
9  }
```

The Java class that comprehends both the test-suites and that is the target of the Gradle

daemon, as we will see in section 5.2.1.1, is the following:

```
1 @Suite
2 @SuiteDisplayName("Unit Test Suite")
3 @SelectClasses(
4     MoviesControllerTests::class,
5     MoviesServiceTests::class
6 )
7 class UnitTestSuite {
8 }
```

4.1.2.2 Google Jib

Jib builds containers without using a Dockerfile [23d]. The adoption of such tool in the context of this research has the following objectives:

- **Faster updates** of the changes made to the microservices. Jib separates the application into multiple layers, splitting dependencies from classes. This eliminates the need to wait for Docker to rebuild the entire Java application, enabling the seamless integration of only the altered layers.
- **Easier integration with Gradle.** Jib seamlessly integrates in the Gradle configuration, allowing for an even more convenient approach in CI/CD.
- **Adopt a daemon-less approach.** Jib reduces the CLI dependencies, building the Docker image from within Maven or Gradle and pushing to any declared registry.

Within the *MoviesCatalog* application, the following configuration has been added to the `build.gradle.kts` file:

```
1 /* --- Google Jib configurations --- */
2 jib {
3     container {
4         args = listOf()
5         ports = listOf("18080/tcp")
6         workingDirectory = "/app"
7     }
8
9     from {
10         image = "amazoncorretto:17-alpine"
11     }
12
13     to {
14         auth {
15             username = System.getenv("CONTAINER_REGISTRY_USERNAME")
16             password = System.getenv("CONTAINER_REGISTRY_PASSWORD")
17         }
18     }
19 }
```

Specifically, the source image from which the application is built is the `corretto` Alpine version from Amazon, a no-cost, production-ready distribution of the Open Java Development Kit (OpenJDK) [23a]. The final image exposes port `18080` and it is pushed to my personal Docker registry for simplicity reasons (`dvdr00t/api-moviescatalog`).

We will see in Chapter 5 how to automatically run Jib builds and how to integrate username and password authentication within CI/CD.

4.1.2.3 OpenAPI Specification file

As discussed in 2.3.2, one of the fundamental pillars of this research is the incorporation of the OpenAPI Specification standard. Gradle allows the seamless integration of the `springdoc-openapi-gradle-plugin`. This enables to use OpenAPI Specification standards in the *MoviesCatalog* application. Specifically, the OpenAPI Specification file is automatically generated in the following process: the executed `forkedSpringBootRun` task runs the application in a background environment while the `generateOpenApiDocs` task, which subsequently starts, makes a REST call to the applications defined URL, downloading and storing the OpenAPI Specification file as YAML or JSON.

A configuration class is provided, which defines a `@Bean` returning an `OpenAPI` object instance:

```
1 @Configuration
2 class OpenAPIConfig {
3     @Bean
4     fun moviesCatalogOpenAPI(): OpenAPI? {
5         return OpenAPI()
6             .info(Info()
7                 .title("moviescatalog-service")
8                 .description("The MoviesCatalog APIs microservice manages
9                 ↪ [...]")
10                .version("v0.0.1")
11                .license(License().name("Apache
12                ↪ 2.0").url("http://springdoc.org"))
13                .contact(Contact()
14                    .name("Davide Arcolini")
15                    .email("d.arcolini@reply.it")
16                )
17            )
18     }
```

In addition to that, custom `components`, `parameters`, `paths`, `security` and `extensions` schemes are provided, in order to adapt the OpenAPI Specification file which is generated to the specific needs of the final application microservice. This fine-tuning enables both an efficient 42Crunch Security Audits scans (which would otherwise complain of the missing features in according to the OWASP API Security) and the generation of valid Kong configuration files. For instance, the following parameter schema is customized to define the properties that the `:name` parameter must possess in the REST calls to the `PATCH` or `DELETE` endpoints of the APIs.

```

1  /* --- custom schemes configurations --- */
2  val nameParameterSchema: Schema<Any> = Schema<Any>()
3      .type("string")
4      .minLength(1)
5      .maxLength(50)
6      .pattern("[A-Za-z0-9\\- ]+$")
7      .additionalProperties(false)

```

In the `build.gradle.kts` file, the configuration of the Gradle plugin is provided, defining the output file name and extension (either JSON or YAML) of the OAS file, the URL at which the Gradle task should make the REST call to retrieve the final document and the maximum timeout that the Gradle daemon should wait before failing the task (especially useful in CI/CD).

```

1  openApi {
2      apiDocsUrl.set("http://localhost:18080/api-docs")
3      outputFileName.set("openapi.yaml")
4      waitTimeInSeconds.set(120)
5  }

```

We will see in section 5.2.2.1 how to automatically manage the generation of OpenAPI Specification file in CI/CD.

4.1.2.4 Performance tests with Gatling

The last plugin adopted in this application is the Gatling Gradle Plugin. The Gradle plugin allows to run Gatling tests from the command line, integrating this testing framework in the application [23c]. The integration between Kotlin and the Gatling Gradle plugin involves defining various entities and structuring performance testing scenarios. Specifically:

- **Simulations.** A simulation is the class defining the behavior and configuration for load testing. It's the core entity where the interactions and load scenarios are specified. Gatling simulations are written in a clean and expressive DSL (Domain-Specific Language), making it easy to define complex user behaviors.

```

1  class MoviesCatalogSimulation: Simulation() {
2
3      /* load simulation configurations */
4      init {
5          setUp(
6              scenario.injectOpen(
7                  nothingFor(5),
8                  rampUsers(USER_COUNT).during(RAMP_DURATION)
9              )
10         ).protocols(httpProtocol)
11     }
12 }

```

- **Scenarios.** A scenario is a part of the simulation that represents a specific user journey

or a series of HTTP requests. Because performing loading tests requires very specific network capabilities and configurations, and this is only a proof of concept, a single and basic scenario has been defined, performing only 3 HTTP requests.

```
1  /* scenario configurations */
2  private val scenario: ScenarioBuilder = scenario("Movies Catalog Load
   ↳ Test")
3      .exec(getAllMovies).pause(2)
4      .exec(getAllMoviesWithFilter).pause(2)
5      .exec(getSingleMovie).pause(2)
```

- **HTTP Requests.** In Gatling, HTTP requests represent the interactions with the application. These requests include the target URL, HTTP method, request headers, and request parameters. Specifically, three user requests have been defined in the simulation:.

```
1  /* API calls configurations */
2  private val getAllMovies: ChainBuilder = exec(
3      http("Get all movies")
4          .get("/movies")
5  )
6  private val getAllMoviesWithFilter: ChainBuilder = exec(
7      http("Get all movies with filter")
8          .get("/movies?genre=Drama")
9  )
10 private val getSingleMovie: ChainBuilder = exec(
11     http("Get a single movie")
12         .get("/movies/Avatar")
13 )
```

- **Simulation Configuration.** Finally, each simulation can be configured with additional various parameters and settings, such as the number of virtual users, ramp-up time, and duration of the test. These settings determine the load and duration of the performance test.

```
1  /* http configurations */
2  private val httpProtocol: HttpProtocolBuilder = http
3      .baseUrl("http://20.103.94.208/")
4      .acceptHeader("application/json")
5      .contentTypeHeader("application/json")
6  private val USER_COUNT: Int = System.getProperty("USER_COUNT",
   ↳ "5").toInt()
7  private val RAMP_DURATION: Long = System.getProperty("RAMP_DURATION",
   ↳ "5").toLong()
```

We will see in [5.3.2.3](#) how to manage the execution of the Gatling performance tests in CI/CD.

4.2 Scenario: manage confidential information

The first scenario discussed in this chapter focuses on the **secure management of confidential information**, like keys and credentials, within Kong microgateway [Kon23l]. Through the adoption of the **API Microgateway Pattern**, the significance of centralized entities for secrets management becomes even more prominent in comparison to traditional API gateways. As this section aims to present, the decentralized management of confidential information is achieved by leveraging the integration of Kong microgateway with **HashiCorp Vault**.

In the realm of Kong microgateway operations, the term *confidential information* pertains to data that must remain undisclosed to unauthorized users. This information is pivotal in the central configuration of APIs. The most commonly employed forms of confidential information that have been used in the presented Kong microgateway scenarios include usernames and passwords for data stores (e.g. employed in conjunction with PostgreSQL), X.509 certificates containing RSA private keys, sensitive fields within plugin configurations, primarily used for purposes like authentication, hashing, signing, encryption and API keys.

Kong Gateway offers the functionality to securely store specific values using a designated **Vault entity**. By choosing to store sensitive data as secrets, it is guaranteed their concealment throughout the platform. This approach ensures that these secrets won't be inadvertently revealed in places like the `kong.conf` file, declarative configuration files, or logs. Instead, it is convenient to refer to each secret through a vault reference, having the form:

```
1 {vault://vault-reference/my-secret-postgres-password}
```

Where `vault` is the scheme (i.e. the Kong protocol standard) that is used to indicate in the configuration that this is a secret, `vault-reference` is the name of the backend managing the secret and `my-secret-postgres-password` corresponds to the secret variable defined.

The following Vault implementations are supported in Kong:

- **AWS Secrets Manager**: can only be used with an Enterprise License.
- **GCP Secrets Manager**: can only be used with an Enterprise License.
- **HashiCorp Vault**: can only be used with an Enterprise License.
- **Environment Variable**: can be used also with Open-Source version of Kong.

In the upcoming sections, the Kubernetes cluster and Kong configurations will be presented.

4.2.1 Cluster infrastructure

This section describes the implementation of the Kubernetes cluster in which the integration between Kong microgateway and HashiCorp Vault has been tested. Specifically, as the HashiCorp Vault integration is limited to the Enterprise version of Kong Gateway, the proof of concept has been developed in a **Azure Kubernetes Service** (AKS) cluster set up within an Azure Subscription provided by **Spike Reply** (DE). The AKS cluster, limited to the considered scenario as shown in 4.3, is composed of the following entities:

- Two **Kubernetes Services**, exposing both the traffic to the *MoviesCatalog* Pod, which does not serve meaningful purposes in this scenario and therefore it will not further

discussed and the traffic to the HashiCorp Vault, which should be limited to the interactions between administrators and the vault servers themselves (*note that: access control is not taken into account, but traffic flowing through this K8s service should be limited and monitored*).

- A **Kubernetes Pod**, containing the *MoviesCatalog* microservice and the **Kong microgateway**, specifically configured as shown in section 4.2.2.1.
- A **Kubernetes Pod**, containing the HashiCorp Vault server in development mode in which secrets have been stored to test the integration with **Kong microgateway**.

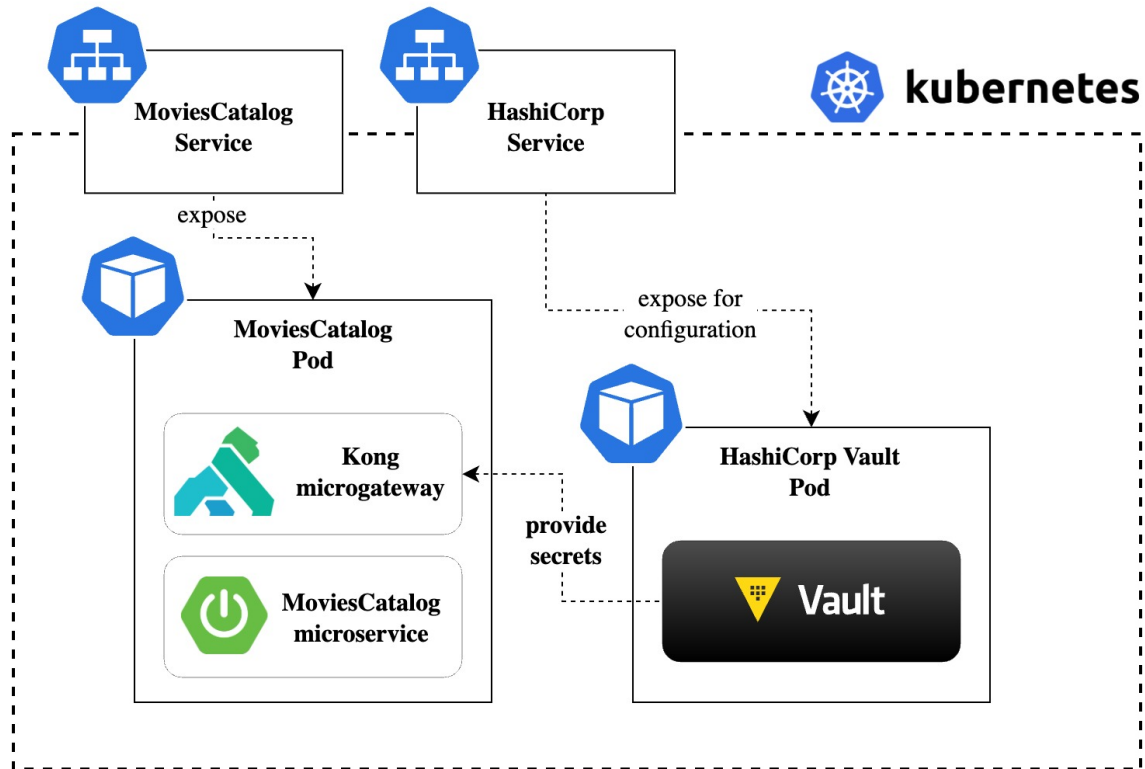


Figure 4.3: Scenario implementing the integration within Kong microgateway and HashiCorp Vault.

4.2.1.1 Configuring HashiCorp Vault

HashiCorp Vault is a robust and versatile tool that plays a pivotal role in the realm of secrets management, data protection, and secure access to sensitive information within modern software systems. It serves as a central repository for securely storing and managing confidential data, ranging from authentication credentials and encryption keys to API tokens and other vital secrets. HashiCorp Vault is designed to offer a secure, organized, and auditable approach for handling secrets in a wide array of applications and environments.

Vault primarily operates with **tokens**, where each token is linked to a client's specific policy. These policies are path-oriented and govern the actions and permissions concerning paths for individual clients. Within the Vault system, tokens can be generated manually and allocated to clients, or clients can acquire tokens by logging in. The fundamental workflow with HashiCorp

Vault is the following:

1. Client submit their identity to perform **authentication**.
2. Vault **validates** the client identity.
3. Client is matched against a Vault security policy to be **authorized**.
4. Vault grants client **access** to the stored secrets, based on the matched policy.

This scenario shows a simple integration of a dev-instance of the HashiCorp Vault server with Kong microgateway. As the integration represents a simple proof of concept, server is deployed on the Kubernetes cluster in **development mode**, without providing any means of **High Availability (HA)** through multiple replicas or even persistence through the integration with an external database (secrets and keys are stored in-memory). The server is installed using [Helm](#) package manger with the following configuration:

```
1 server:
2   dev:
3     enabled: true
4     devRootToken: "securetoken"
```

Here, the `devRootToken` parameter is used to define an Access Token that can be used to access the Vault with admin privileges. This is a confidential piece of information and should not be exposed to malicious or untrusted users.

4.2.1.2 Storing client secrets in HashiCorp Vault

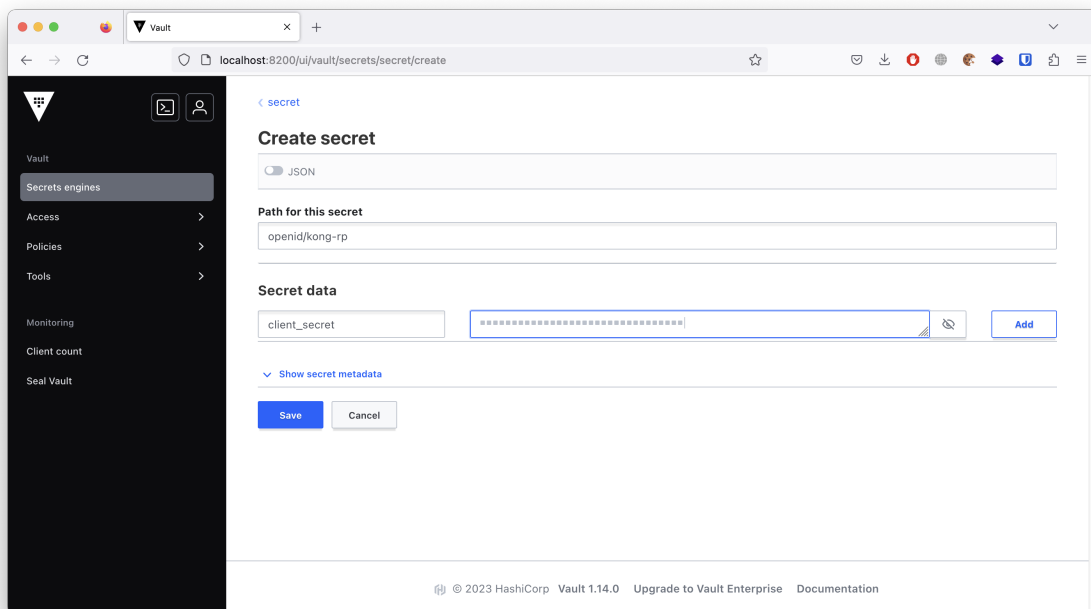


Figure 4.4: HashiCorp Vault User Interface (UI) to add a new secret.

Once the development server is installed inside the cluster, developers can access the Vault UI through the Kubernetes Service managing the HashiCorp Vault Pod to add new secrets for the swarm of Kong microgateways that they intend to deploy. Inside the UI, as shown in 4.4, new secrets can be added which will be later accessible by all Kong microgateways possessing the `devRootToken` previously defined.

Once the secret is ready, it is possible to configure Kong microgateway to access the Vault and retrieve the confidential information.

4.2.2 Kong microgateway

Allowing Kong microgateway to interact with the HashiCorp Vault server in the Kubernetes cluster requires the configuration of a **Vault entity**.

4.2.2.1 Configuring Vault entity

A basic example of Vault entity configuration is the following:

```
1 vaults:
2 - description: Storing secrets in HashiCorp Vault
3   name: hcv
4   prefix: hashicorp-vault
5   config:
6     host: vault.microgateway
7     namespace: microgateway
8     kv: v2
9     mount: secret
10    port: 8200
11    protocol: http
12    token: secretoken
```

Where the `host` parameter indicates the address of the Vault server in the `microgateway` namespace, reachable at `port 8200`. The `token` parameter holds the value of the token that grants the authentication and authorization of Kong microgateway to the Vault server. Two important considerations must be done here:

1. The `token` value **must not** be left in plain sight in the Kong configuration file. As presented in this research, the `kong.yaml` file is managed in pipeline from a CI/CD server: let the server manage and inject the secret when the file is configured. For instance, in a GitHub Action workflow:

```
1 - name: Setup Kong Vault Access Token
2   id: configure-token
3   env:
4     KONG_VAULT_TOKEN_SECRET: ${ secrets.KONG_VAULT_TOKEN }
5   run: |
6     KONG_VAULT_TOKEN=$KONG_VAULT_TOKEN_SECRET
7     sed -i "s/TOKEN_PLACEHOLDER/$KONG_VAULT_TOKEN/g" kong-plugins.yaml
```

2. The `token` value used here refers to the `DevRoot` Access Token defined in the deploy-

ment of the Vault server. Supposedly, each Kong microgateway should not be granted admin permission on the Vault server and more fine-grained policies for authentication and authorization should be setup. Each developers team should access only a restricted section of the HC Vault, following the **Least Privilege Access** principle.

Once the Vault entity is in place, it is possible to reference the secret inside any Kong configuration file. For instance, let's consider the OpenID Connect plugin, which requires the definition of the `client_secret` parameter. As the `kong-plugin.yaml` file is stored under Version Control on a remote Git repository, the parameter should not be kept in plain sight. Rather, it is convenient to access it through the HashiCorp Vault as follows:

```
1 plugins:
2   - name: openid-connect
3     config:
4       issuer: http://keycloak.microgateway:8080/realms/moviescatalog
5       client_id: [kong-rp]
6       client_secret: [{"vault://hashicorp-vault/openid/kong-rp/client_secret}"]
```

4.3 Scenario: integrate observability

The second scenario explored in this chapter delves into the integration of **monitoring capabilities** within a cluster of Kong microgateways. In this scenario, the focus is set on the establishment of a robust monitoring system to ensure the health, performance, and operational efficiency of the deployed microgateways. By adopting the **API Microgateway Pattern**, it becomes crucial to centralize the collection of data in external entities, ensuring that each deployed microgateways can be scraped and metrics from each instance can be grouped together.

Kong Gateway offers a seamless integration with Prometheus through the **Prometheus Plugin**, which exposes metrics related to Kong and proxies upstream services in the Prometheus exposition format, in order to be scraped by a Prometheus Server [Kon23j]. Moreover, because Kong scraped metrics should be properly displayed to efficiently monitor the status of the cluster and the traffic flowing through the Kong microgateways, **Grafana** queries the data from the Prometheus server to build specific dashboards and display information.

In the upcoming sections, the Kubernetes cluster and Kong configurations will be presented.

4.3.1 Cluster infrastructure

This section describes the implementation of the Kubernetes cluster in which the integration between Kong microgateway, Prometheus and Grafana has been tested.

The Kubernetes server, limited to the considered scenario as shown in 4.5, is composed of the following entities:

- Three **Kubernetes Services**, exposing:
 - the traffic to the *MoviesCatalog* Pod, which does not serve meaningful purposes in this scenario and therefore it will not further discussed;
 - the traffic to the Prometheus Server, which should be limited to the interactions

between administrators and the server itself (*note that*: access control is not taken into account, but traffic flowing through this K8s service should be limited and monitored);

- the traffic to the Grafana server, to expose dashboards and monitor the status of the cluster (*note that*: access control is not taken into account, but traffic flowing through this K8s service should be limited and monitored).
- A **Kubernetes Pod**, containing the *MoviesCatalog* microservice and the **Kong microgateway**, specifically configured to use the **Prometheus plugin**, as shown in section 4.3.2.1.
- A **Kubernetes Pod**, containing the **Prometheus server** used to scrape metrics from the Kong microgateway.
- A **Kubernetes Pod**, containing the **Grafana server** used to collect data from Prometheus and expose dashboards to the administrators of the cluster.

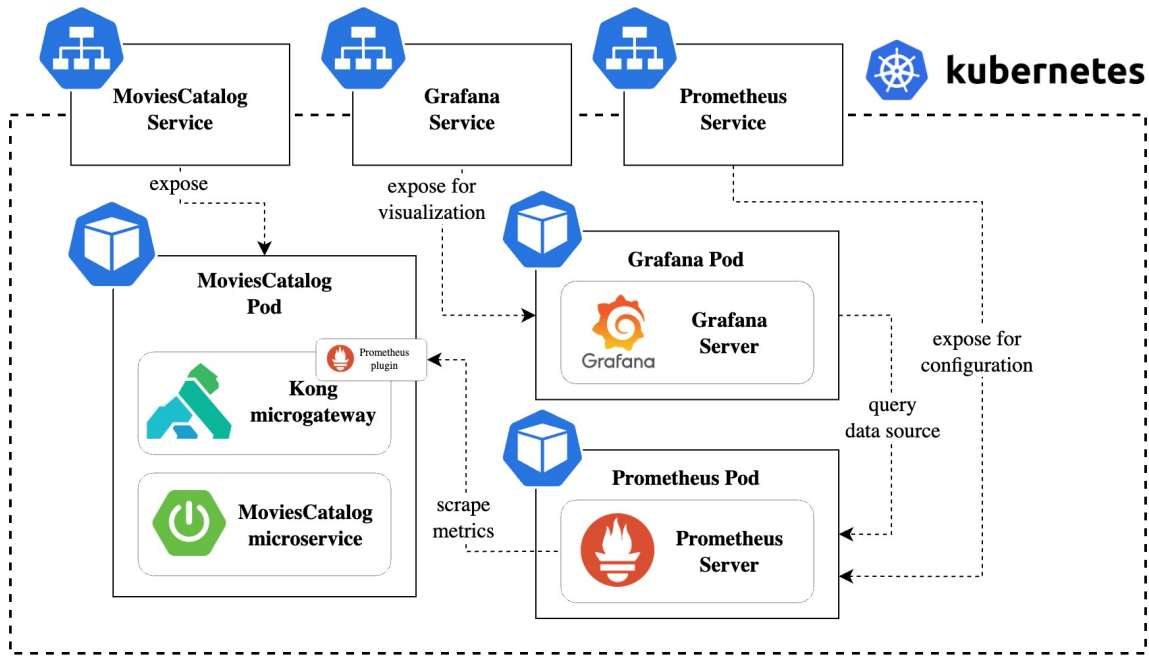


Figure 4.5: Scenario implementing the integration within Kong microgateway, Prometheus and Grafana.

4.3.1.1 Configuring Prometheus

Prometheus is an open-source monitoring system with multiple features, such as: **dimensional data model** with time series database, a custom query language, and specific alerting mechanism. Prometheus gathers and stores metrics in the form of time series data, meaning that metric information is stored with the timestamp of its recording, along with labels, i.e., optional key-value pairs.

Metrics, which are numerical measurements capturing various aspects, are essential for understanding an application’s behavior. These metrics are often organized as time series data,

showing how values change over time. Depending on the application, developers and administrators may be interested in measuring different factors. While it could be request times, active connections or queries, metrics serve as valuable tools for diagnosing issues and optimizing application performance. Within the **API Microgateway Pattern**, collecting metrics from multiple Kong microgateways enable a complete overview of the status of the cluster.

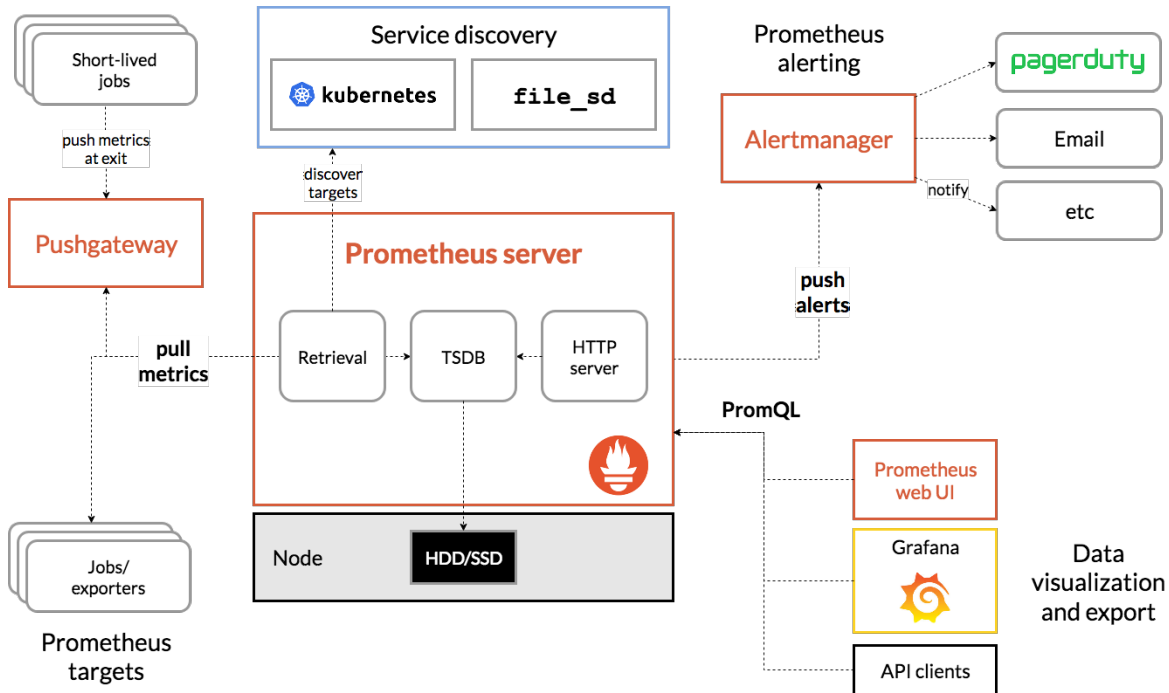


Figure 4.6: Prometheus architecture [Source [Prometheus](#)].

Although Prometheus has a very comprehensive architecture, as depicted in figure 4.6, for the purpose of this proof of concept only the exporters pulling metrics from Kong microgateway instances are relevant. Indeed, the focus is on the swarm of microgateways and not on the configuration of alerts or service discovery capabilities.

Prometheus server (*note that no High Availability is provided in the proof of concept*) is installed using [Helm](#) package manger with the following configurations:

- **Time and evaluation intervals** that occurs between two consecutive metrics scrape and timeouts.

```

1 server:
2   global:
3     scrape_interval: 10s
4     scrape_timeout: 10s
5     evaluation_interval: 1m

```

- **Target** Kong microgateways that need to be scraped. Here, the `targets` parameter is an array of IPs and ports locating Kong microgateways inside the cluster. While IPs are managed by Kubernetes, port `8001` is the one exposing Status APIs in Kong microgateways, as seen in 4.3.2.1.

```

1 serverFiles:
2   prometheus.yml:
3     scrape_configs:
4       - job_name: kong
5         static_configs:
6           - targets:
7             - 10.0.55.177:8001

```

- **Disabled entities** for this proof of concepts, in order to streamline the deployment process, focusing on essential monitoring aspects without the added complexity.

```

1 alertmanager:
2   enabled: false
3 kube-state-metrics:
4   enabled: false
5 prometheus-node-exporter:
6   enabled: false
7 prometheus-pushgateway:
8   enabled: false

```

In this case, a Prometheus job named `kong` is defined to scrape Kong microgateways endpoints every 10 seconds intervals.

4.3.1.2 Configuring Grafana

Grafana is an open-source platform designed for observability and monitoring, offering **robust visualization and analytics** capabilities. In the context of this research, Grafana serves as an integral component of the monitoring setup alongside Prometheus and Kong microgateways.

Grafana server (*note that no High Availability is provided in the proof of concept*) is installed using **Helm** package manger with the following configurations:

- The Prometheus **DataSource** from which Grafana will query time data series to be displayed in the dashboard. The location of the Prometheus server in the cluster is provided.

```

1 datasources:
2   datasources.yaml:
3     apiVersion: 1
4     datasources:
5       - name: Prometheus
6         type: prometheus
7         url: http://prometheus-server.microgateway.svc.cluster.local

```

- The **Dashboard** along with the **DashboardProvider**. Specifically, the **Kong (official)** Grafana dashboard is installed, with ID `7424`. In addition to the official dashboard, we will see in 6 that also a custom dashboard has been defined to provide more insights of the Kong microgateway cluster.

```

1 dashboardProviders:
2   dashboardproviders.yaml:
3     apiVersion: 1
4     providers:
5       - name: default
6         orgId: 1
7         folder: Prometheus
8         type: file
9         disableDeletion: false
10        editable: true
11        options:
12          path: /var/lib/grafana/dashboards/default
13 dashboards:
14   default:
15     kong-dash:
16       gnetId: 7424
17       revision: 5
18       datasource: Prometheus

```

Once the Grafana server is ready, administrators and developers can access the web application to visualize the status of the Kong microgateways cluster. Figure 4.7 shows one of the page in the [official Kong Grafana dashboard](#), reporting metrics such as globally, per-service and per-route latency, request time, upstream time, total bandwidth, number of requests per seconds (RPS), globally and by returned status code, memory usage by each Kong microgateway and more.

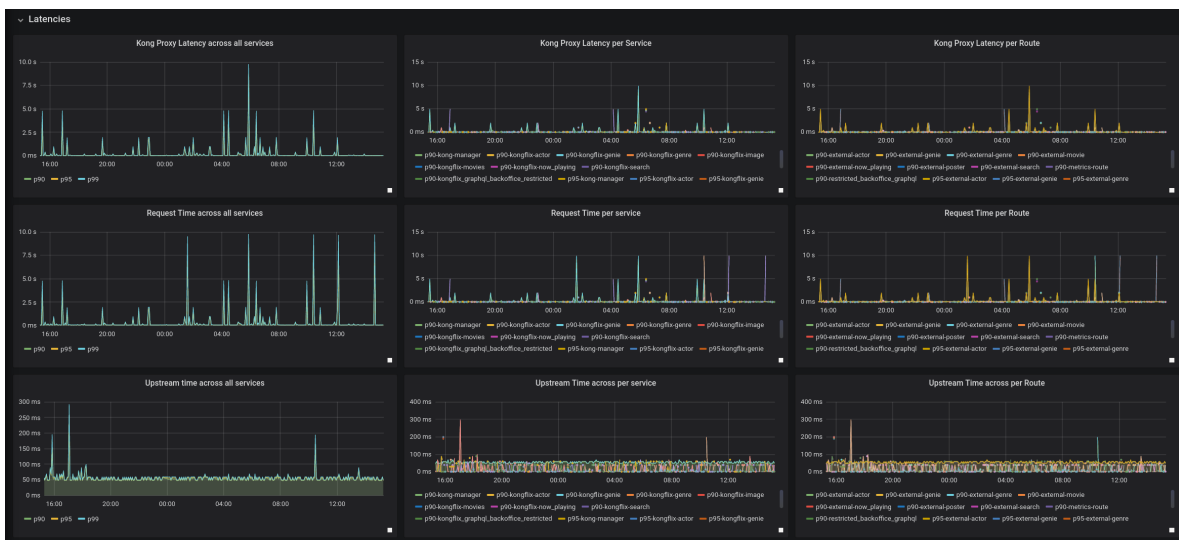


Figure 4.7: Kong (Official) Grafana Dashboard [Source [Grafana](#)].

4.3.2 Kong microgateway

Allowing Prometheus to discover, connect and scrape metrics from each Kong microgateway in the Kubernetes cluster requires the configuration of the **Prometheus plugin** for each entity.

4.3.2.1 Configuring Prometheus plugin

The **Prometheus plugin** facilitates the exposure of metrics associated with Kong and proxied upstream services in the Prometheus exposition format, enabling their collection by a Prometheus Server. These metrics can be accessed through the Admin API and Status API, typically located at the `http://localhost:port/metrics` endpoint. It's important to note that the plugin operates at the node level, necessitating Prometheus to discover all Kong microgateways via a service discovery mechanism (in the presented case, Kong microgateways endpoints are hard-coded in the Helm configuration file) and fetch data from the configured `/metrics` endpoint of each node.

An example of configuration for the Prometheus plugin, in which all metrics are enabled, is the following:

```
1 plugins:
2   - name: prometheus
3     enabled: true
4     config:
5       per_consumer: true
6       status_code_metrics: true
7       latency_metrics: true
8       bandwidth_metrics: true
9       upstream_health_metrics: true
```

4.4 Scenario: implement Identity and Access Management (IAM)

The third and last section of this chapter will focus on examining various scenarios designed to assess the integration of authentication and authorization features within Kong microgateway. More specifically, multiple use-cases will be explored to evaluate the diverse authentication and authorization flows that encompass the realm of **Identity and Access Management (IAM)** and are currently regarded as industry best practices.

Identity and Access Management, often abbreviated as IAM, affords control over the validation of consumers and their access to the digital resources, such as APIs. The IAM frameworks guarantee that digital resources are accessed by the right individuals at the right time and for the appropriate reasons [Aut23]. In the realm of Identity and Access Management, the two core concepts are:

- **Authentication.** It determines whether users are who they claim to be. It typically encompasses one or more challenges, such as providing valid credentials, supplying biometric data, or answering specific questions. Results of this evaluation are carried in **ID Tokens**, generally governed by the **OpenID Connect** (OIDC) protocol.
- **Authorization.** It determines what users can and cannot do. It is typically enforced through policies and rules, based on information transmitted within **Access Tokens**, generally governed by the **OAuth 2.0** framework.

The **API Microgateway Pattern** embraces the deployment of Kong microgateways in its DB-less configuration, prioritizing lightweight and rapid operations. Tailored Identity and Access Management infrastructures backing up swarm of Kong microgateways were developed to address the need for securely exposing APIs on the internet, for managing consumers

authentication and for enforcing request authorization.

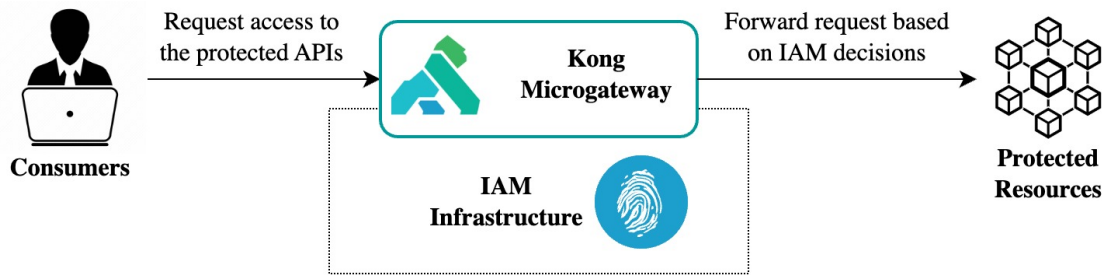


Figure 4.8: An Identity and Access Management (IAM) infrastructure has been developed in these PoC to test Kong microgateway capabilities to implement authentication and authorization flows.

Identity and Access Management is not a ready-made set of rules or protocols that can be directly implemented; instead, it represents a discipline and a framework designed to address the issue of secure access to digital resources. Because of this, in the context of the developed scenarios, the most common and industry recognized frameworks have been integrated:

- **OAuth 2.0.** The OAuth 2.0 authorization framework serves as a **delegation protocol for API access** and represents the widely accepted industry-standard for Identity and Access Management. This open authorization protocol empowers applications to access resources from other web applications without the need to disclose the user's credentials. OAuth 2.0 forms the foundation for enabling third-party developers to leverage major social platforms, including Facebook, Google, and Twitter, for user authentication.
- **OpenID Connect.** The OpenID Connect (OIDC) protocol functions as a straightforward **identity layer built on top of the OAuth 2.0 framework**, simplifying the verification of a user's identity and the retrieval of fundamental profile details from the identity provider.
- **SAML 2.0.** Security Assertion Markup Language (SAML) is an open-standard, XML-based data format facilitating the **exchange of user authentication and authorization** information between businesses and their partner companies or enterprise applications used by their employees.

Most of the considered scenarios that will be presented in this section are carried out in a tailored Kubernetes infrastructure built upon the following open-source technologies:

- **Keycloak.** Keycloak is an **open-source Identity and Access Management platform** providing user federation, strong authentication, user management, and fine-grained authorization. Keycloak has been adopted as Authorization Server (AS) in OAuth 2.0 flows and Identity Provider (IdP) in OpenID Connect and SAML 2.0 flows.
- **Open Policy Agent.** OPA provides an **unified toolset and framework for policy** across the cloud native stack. OPA has been adopted as the Policy Enforcement Point (PEP) in the infrastructure, defining specific policies for each microservices covered by the Kong microgateways.

In the upcoming sections, the Kubernetes cluster and Kong configurations will be presented.

4.4.1 Cluster infrastructure

This section describes the configuration of the Kubernetes cluster in which the integration between Kong microgateway, Keycloak and Open Policy Agent has been tested to implement the Identity and Access Management infrastructure that secures the exposure of the APIs. Because some of the Kong plugins integration is limited to the Enterprise version of Kong Gateway, this proof of concept has been developed both on the **microk8s** (Kubernetes) cluster established within the servers of **Liquid Reply** (IT) and on the **Azure Kubernetes Service** (AKS) cluster set up within an Azure Subscription provided by **Spike Reply** (DE). This serves also the purpose of showcasing how the proposed solutions can be applied both to an on-premise and on-cloud Kubernetes cluster.

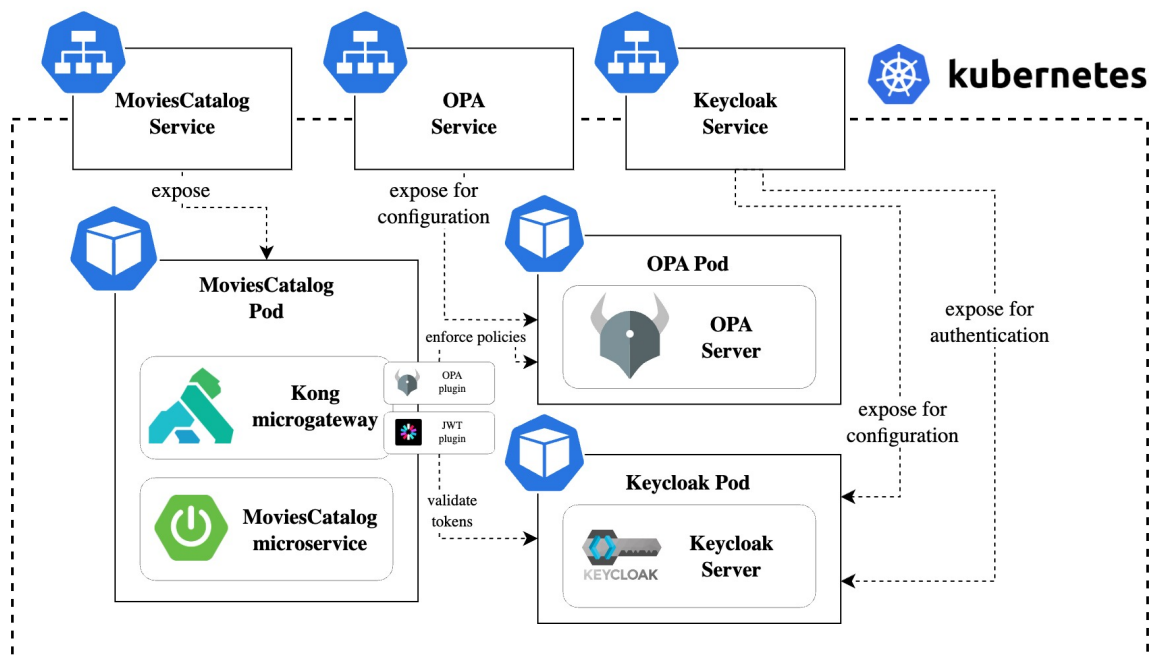


Figure 4.9: Scenario implementing the integration within Kong microgateway, Keycloak and Open Policy Agent.

The Kubernetes cluster, limited to some of the considered scenario as shown in 4.9, is composed of the following entities:

- Three **Kubernetes Services**, exposing:
 - the traffic to the *MoviesCatalog* Pod, which does not serve meaningful purposes in this scenario and therefore it will not further discussed;
 - the traffic to the Open Policy Agent Server, which should be limited to the interactions between administrators and the server itself (*note that*: access control is not taken into account, but traffic flowing through this K8s service should be limited and monitored);

- the traffic to the Keycloak server, to expose configuration and authentication endpoints as specified by the specific flows (*note that*: access control is not taken into account, but traffic flowing through this K8s service should be limited and monitored).
- A **Kubernetes Pod**, containing the *MoviesCatalog* microservice and the **Kong microgateway**, specifically configured to use the specific plugin(s) that correspond to the considered scenario.
- A **Kubernetes Pod**, containing the **Open Policy Agent server** used to enforce authorization based on the loaded policies.
- A **Kubernetes Pod**, containing the **Keycloak server** used to authenticate users and issue Tokens and Assertions valid in the specific considered scenario.

4.4.1.1 Configuring Keycloak

Keycloak is an **open-source Identity and Access Management platform** providing user federation, strong authentication, user management, and fine-grained authorization [Key23]. Keycloak has been adopted as the Authorization Server (AS) in the OAuth 2.0 flows and as the Identity Provider (IdP) in the OpenID Connect and SAML 2.0 flows.

A `quay.io/keycloak/keycloak:latest` Keycloak server has been deployed onto the Kubernetes cluster and started in development mode by using the following Kubernetes Manifest (only the configuration of the container is reported here):

```

1  containers:
2  - name: keycloak
3    image: quay.io/keycloak/keycloak:latest
4    args: ["start-dev"]
5    env:
6      - name: KEYCLOAK_ADMIN
7        valueFrom:
8          secretKeyRef:
9            name: keycloak-secret
10           key: KEYCLOAK_ADMIN_USERNAME
11     - name: KEYCLOAK_ADMIN_PASSWORD
12       valueFrom:
13         secretKeyRef:
14           name: keycloak-secret
15           key: KEYCLOAK_ADMIN_PASSWORD
16   ports:
17     - name: http
18       containerPort: 8080
19   readinessProbe:
20     httpGet:
21       path: /realms/master
22       port: 8080

```

Username and passwords are provided as a Kubernetes secret, and are used to access the Master Realm in Keycloak:

```

1 apiVersion: v1
2 kind: Secret
3 metadata:
4   name: keycloak-secret
5   namespace: microgateway
6 type: Opaque
7 data:
8   KEYCLOAK_ADMIN_PASSWORD: c2VjdXJlcGFzc3dvcmQ=
9   KEYCLOAK_ADMIN_USERNAME: YWRtaW4=

```

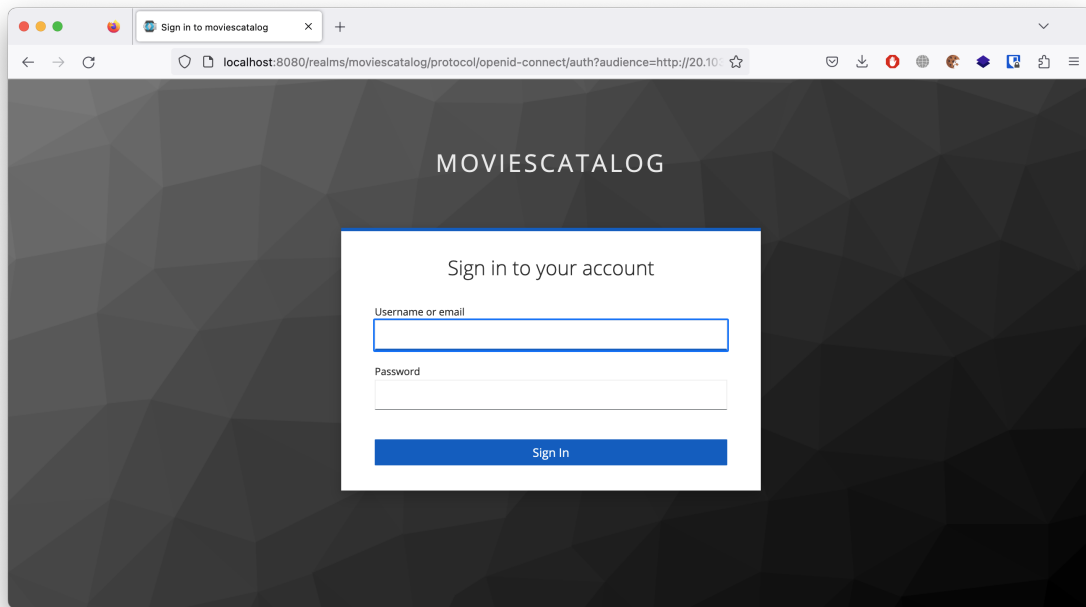


Figure 4.10: Keycloak Single Sign-On (SSO) interface for granting access to Client applications.

Once the server has been deployed, a new Realm has been created, called *MoviesCatalog* in which the specific Client configurations needed for the scenarios and the Users required to grant access to the APIs have been added. Among the large set of Keycloak defined resources, the most relevant to the aim of these scenarios are the following:

- **Clients.** Typically, Clients refer to applications and services that engage with Keycloak for user authentication. Their main objective is to ensure their own security and provide a unified single sign-on solution. Clients can also encompass entities interested in acquiring identity information or an access token to securely access other network services protected by Keycloak.
- **Client Scopes.** When registering a client, it is necessary to specify protocol mappers and role scope mappings for that particular client. Additionally, it is often beneficial to establish a client scope, simplifying the process of creating new clients by sharing certain common configurations. Client scopes also prove valuable when requesting specific claims

or roles to be conditionally determined based on the scope parameter's value. Keycloak introduces the concept of a client scope to address these requirements.

- **Users.** Users are individuals with the capability to access the APIs by logging in. They can possess various attributes linked to them, such as email, username, address, phone number, and birthday. Additionally, users can be assigned to specific groups and granted particular roles within the system.
- **Access Tokens.** The token, which can be included in an HTTP request to authorize access to the service being called, is a fundamental component of the OpenID Connect and OAuth 2.0 specifications.

Details on how to configure Clients, Client Scopes, Users, Access Tokens and the other Keycloak resources are presented in section 4.4.3, as they are specific to the consider scenario. Figure 4.10 shows the Keycloak Single Sign-On (SSO) interface that is prompted to the Users when it is required to perform log in and grant Client applications access to the protected APIs.

4.4.1.2 Configuring Open Policy Agent

Open Policy Agent (OPA) is an open source, general-purpose policy engine that unifies policy enforcement across the stack [Age23b]. OPA offers a **high-level declarative language** enabling **policy specification as code**, along with straightforward APIs for shifting policy decision-making away from the applications. It can be effectively used to enforce policies in diverse contexts, including microservices, Kubernetes, CI/CD pipelines, API gateways, and beyond. Indeed, in the following scenarios, OPA has been adopted as the Policy Enforcement Point (PEP) in the infrastructure, defining specific policies for each microservices covered by the Kong microgateways.

OPA separates the process of policy decision-making from policy enforcement. In scenarios where Kong microgateway requires policy decisions, it consults OPA and provides structured data for evaluation, which in the considered cases corresponds to the Signed JSON Web Token (JWS) issued by Keycloak, as input. The, OPA generates policy decisions by evaluating the query input against the defined policies and the provided data. The model features well in the **API Microgateway Pattern**, as it enables the externalization of the policy enforcement practices from the cluster of Kong microgateways.

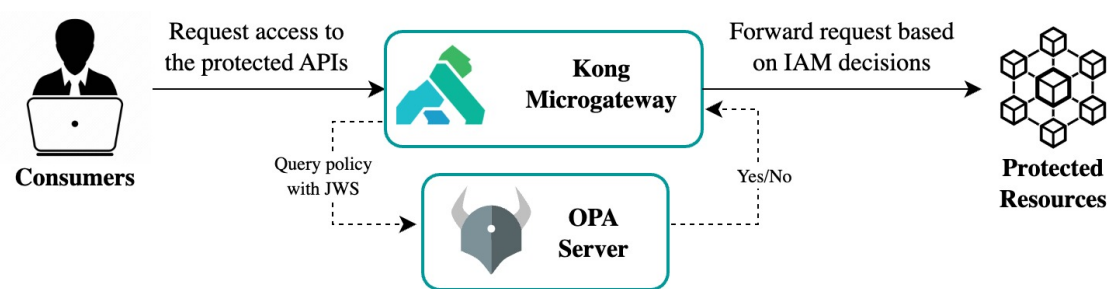


Figure 4.11: Enforcement of policies with OPA.

A `openpolicyagent/opa:0.56.0` Open Policy Agent server has been deployed onto the Kuber-

netes cluster and started in development mode by using the following Kubernetes Manifest (only the configuration of the container is reported here):

```
1 containers:
2 - name: opa
3   image: openpolicyagent/opa:0.56.0
4   ports:
5   - name: http
6     containerPort: 8181
7   args:
8   - "run"
9   - "--ignore=.*" # exclude hidden dirs created by Kubernetes
10  - "--server"
11  - "/policies"
12  volumeMounts:
13  - readOnly: true
14    mountPath: /policies
15    name: example-policy
```

Policies are scripted in Rego, a language that supports structured document models like JSON. Rego queries function as assertions on the data stored in OPA. These queries are instrumental in defining policies that enumerate instances of data deviating from the anticipated state of the system. In particular, all the considered scenarios that employ OPA as the Policy Enforcement Point define a policy containing the following set of rules (*note that this policy is defined to test the architecture. It is not intended to be a real-case scenario policy, in which several more considerations must be taken into account*):

- `valid_issuer`: the issuer of the token must be registered in the OPA server. This allows to discard tokens that are not generated from the specified Authorization Server.
- `not_expired`: the token must still be valid and not expired.
- `valid_method` and `valid_path`: the request must be a `POST` request to the the `/movies` endpoint.
- `is_scoped`: the input token must have, among its scopes, the `read-movie` scope.

The resulting and complete policy is the following.

```
1 package readmovies
2 default allow = false
3 issuers := ["http://localhost:8080/realms/moviescatalog"]
4
5 decode_token(header) = payload {
6   [_ , token] = split(header, " ")
7   [_ , payload, _] = io.jwt.decode(token)
8 }
9
10 allow {
11   decoded_token :=
12     ↪ decode_token(input.request.http.headers["authorization"])
```

```

12     valid_issuer(decoded_token)
13     not_expired(decoded_token)
14     valid_method
15     valid_path
16     is_scoped(decoded_token)
17 }
18
19 valid_issuer(payload) { payload.iss = issuers[_] }
20 not_expired(payload) {
21     now := time.now_ns() / 1000000000
22     now < payload.exp
23 }
24 valid_method { input.request.http.method == "GET" }
25 valid_path { startswith(input.request.http.path, "/movies") }
26 is_scoped(payload) { split(payload.scope, " ")[_] == "read-movie" }

```

The Policy REST API exposes CRUD endpoints for managing policy modules [Age23a]. Policy modules can be added, removed, and modified at any time. Specifically, to add the new policy OPA exposes the `PUT /v1/policies` API that creates or updates the policy module. If the policy module does not exist, it is created. If the policy module already exists, it is replaced:

```

1 PUT /v1/policies/readmovie HTTP/1.1
2 Host: localhost:8181
3 Content-Type: text/plain
4 Content-Length: <length>
5
6 <policy>

```

4.4.2 Kong microgateway

The following list comprehends the Kong plugins adopted to set up the Identity and Access Management infrastructure, allowing Kong microgateway to interact with the Keycloak server and the Open Policy Agent server.

- **JWT**. The JWT plugin provides the capability to validate requests containing **JSON Web Tokens** (JWTs) signed with either HS256 or RS256, adhering to RFC 7519 standards. This plugin serves as a straightforward yet highly adaptable tool, enabling Keycloak, acting as an Authorization Server, to generate JWTs that will subsequently undergo validation by Kong.
- **OPA** [*Enterprise only*]. The OPA (**Open Policy Agent**) plugin is in charge of forwarding the request to the OPA server, querying data to evaluate the input against the defined policies.
- **OpenID Connect** [*Enterprise only*]. The OpenID Connect (OIDC) plugin enriches Kong with OIDC capabilities, overseeing not only Access Tokens and Refresh Tokens but also the management of **ID Tokens**. It serves as a seamless expansion of the JWT plugin and can be employed as an alternative, enabling the fulfillment of OIDC functionalities.

- **SAML** [*Enterprise only*]. The SAML plugin in Kong serves a critical role within the **Security Assertion Markup Language** (SAML) framework, an open standard designed to facilitate the exchange of authN/authZ data between identity providers (IdP) and a service provider (SP). Kong SAML plugin functions as the service provider, primarily tasked with initiating a login to the identity provider, which is referred to as an **SP-Initiated Login**.

In the upcoming sections, all the snippets of code used to configure the plugins are presented. As a general consideration, confidential information like keys, secrets and certificates are displayed in plain sight for clearness. However, in a real use-case, each private data should be stored in a Vault as defined in 4.2.2.1.

4.4.2.1 Configuring JWT plugin

Configuring the JWT plugin is straightforward. The first step is to create a **Consumer entity** in Kong. As different consumers can have different keys and secrets, this will allow Kong to map the received token to a specific consumer and verify the signature of that token. Specifically, Kong will use the `key` value of the consumer to perform the mapping with the specified property in the plugin configurations. In the context of Kong microgateway, a Consumer is mapped to the Client application registered in Keycloak.

This section presents a general example of configuration, which has then been adapted to each use-case described in 4.4.3.

```

1 consumers:
2   - username: insomnia
3     jwt_secrets:
4       - algorithm: RS256
5         key: BSo_b184Dn9XWei7jL0MT728aN_WuLLEhUjv7rQrgLE
6         secret: dummy-value
7         rsa_public_key: |
8             -----BEGIN PUBLIC KEY-----
9             MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA3W0or00VAqNhjGKkOrrE
10            8IIwawISn86XSzWXrqYYQg/5n6As9oNnUTKzswgQDyMy9MOjxup+yFTRVxdxFrZB
11            0+q/noYmbfP7jRSn8248S5JqxsqoPOJUy/reG2iIaVu/Jp7/voJYAKq8XwtX5nvZ
12            JDCXzUDXvcITyPxnxeHQ8PB6iUSfTuDNQs9q9ipQb09EmSqsPtjqtOYVvKZK1C429
13            TmnUAAZyIDg6AbUXbYcn+1IM41NqDQv3ASRDctPYp3QnLUDDICfOPW16Czhmy+Mt
14            WhrZxsg0h+KSTdJfYX7tP2pNRm2Pz0bubtE9qrbio43qB5npBY/nU4b0wht/tYt2
15            NwIDAQAB
16            -----END PUBLIC KEY-----

```

Specifically, in the presented configuration the Consumer is `insomnia`, indicating that the Insomnia desktop application will interact with the APIs and therefore a new set of credentials is added to such Consumer. The `algorithm` property in the `jwt_secrets` object specifies how the Access Token should be verified. Specifically, a RS256 signing algorithm is used, with `rsa_public_key` being the one specified later (it can be obtained in Keycloak Realm configuration). Finally, `secret` is a dummy value that must be added in the case the algorithm is not symmetric. In case of a symmetric algorithm is used, `rsa_public_key` should be defined with a dummy value.

Once the consumer has been created, it is possible to add the plugin to the `kong-plugins.yaml`.

The configuration is the following:

```
1 plugins:
2 - name: jwt
3   route: moviescatalog-service_createmovie
4   enabled: true
5   config:
6     key_claim_name: kid
7     secret_is_base64: false
8     claims_to_verify:
9     - exp
```

In this case, the plugin applies only to the `CreateMovie` route, which is mapped to the `POST /movies` API as previously indicated. The plugin is `enabled` and uses the `kid` identifier of the token to match consumers' keys, as previously mentioned. The secret (in this case, the public key) is not base64-encoded. The only claim to be verified is the token expiration (which is also check by OPA, just for testing purposes).

4.4.2.2 Configuring OPA plugin

Configuring the OPA plugin is even more straightforward. Once the OPA server is in place, it is necessary to add the `opa` plugin to the list of plugins, indicate where the OPA server is running and which rule to apply.

```
1 plugins:
2 - name: opa
3   route: moviescatalog-dbless-service_createmovie
4   enabled: true
5   config:
6     opa_host: 10.0.101.149
7     opa_port: 8181
8     opa_path: "/v1/data/createmovie/allow"
9     ssl_verify: false
```

The plugin is applied, only to the `CreateMovie` route. It is `enabled` and the OPA server is running in the cluster at `10.0.101.149` with port `8181`. The rule is inside the `createmovie` package, as defined in [4.4.1.2](#). It is accessible by making query requests to the API endpoint defined as `v1/data/createmovie/:rule`, in this case: `/v1/data/createmovie/allow`. The SSL verification is skipped and the other properties indicate what Kong should forward to the OPA server.

4.4.2.3 Configuring OIDC plugin

As stated in the official OpenID Connect Kong plugin documentation [[Kon23i](#)]: *"this plugin can be used to implement Kong as a (proxying) OAuth 2.0 resource server (RS) and/or as an **OpenID Connect relying party (RP)** between the client, and the upstream service"*. In this scenario, with respect to the Resource Server, **Kong microgateway will be representing the OpenID Connect Relying Party**. Kong will entirely manage the authentication flow, requesting the exchange of Authorization Codes for Access Tokens and **Identity Tokens (ID Tokens)**, that can be used respectively to authorize the request with OPA and to run

introspection for additional user information.

This plugin has a lot more configuration parameters for a very fine-grained customization. Indeed, the OpenID Connect plugin provides for a variety of different authentication flows that can be read in the official documentation. Still, this section provides a comprehensive and detailed guide to configure Kong using the OAuth 2.0 Authorization Code Flow with OpenID Connect capabilities. The required configuration are presented in the following list:

- At first, Kong microgateway must know **where the OpenID Connect provider is**, in this case, Keycloak. The Kong parameter holding this reference is the `issuer` parameter. `issuer`, which is the only mandatory parameter, tells the plugin where to find the issuer in the network.
- Next, Kong microgateway must authenticate with Keycloak too. It must know **how to do it** and it has many ways to do so: using the client secret passed as header parameter (`client_secret_basic`) or in the body of the request (`client_secret_post`), sending client assertion signed with the client secret as part of the body (`client_secret_jwt`), sending client assertion signed with the private key of the client as part of the body (`private_key_jwt`) or even not authenticating (`none`). As Keycloak will be configured to accept Clients authentication in `POST` requests, the required value will be `client_secret_post`.
- Once Kong microgateway knows how to authenticate, it must **hold the credentials** to pass authentication. Specifically, `client_id` (i.e., the client id(s) that the plugin uses when it calls authenticated endpoints on the identity provider) and `client_secret` (i.e., the client secret).
- At this point Kong microgateway knows where to authenticate, how and with which credentials. It is necessary to specify **which authentication flow** will be used by the Client. In the considered scenario, the OIDC plugin uses the `authorization_code` flow.
- Finally, as this is an OpenID Connect flow, it is necessary to request Keycloak the `openid` scope when issuing the Access Token, so that it can also **issue the ID Token**. `scopes` holds the scopes passed to the authorization and token endpoints.

With that in mind, the final configuration is the following.

```
1 plugins:
2   - name: openid-connect
3     route: moviescatalog-service_retrieveallmovies
4     enabled: true
5     config:
6       issuer: http://keycloak.microgateway:8080/realms/moviescatalog
7       client_id:
8         - kong-rp
9       client_secret:
10        - v0HZF7tDqgzJXWE3U2tcoYJLrPFyJg7S
11      scopes:
12        - create-movie
13        - openid
14      auth_methods:
```

```
15     - authorization_code
16     client_auth:
17     - client_secret_post
18     search_user_info: true
```

Some considerations:

- Inside `scopes`, the `create-movie` scope has been added. This is done to grant Kong microgateway the access to the OPA-protected resource.
- The `search_user_info` set to `true` indicates Kong microgateway to introspect for additional user information when requesting the Access Token.
- The `client_secret` is defined in plain-sight. As the `kong-plugins.yaml` file is kept under version control, it will be pushed in the remote repository and it must be considered as a confidential piece of information. Indeed, this parameter is defined as **referenceable**, meaning that it can be obtained from third-party encrypted storage like Vaults.

4.4.2.4 Configuring SAML plugin

As stated in the official SAML plugin documentation [Kon23m], the Kong SAML plugin represents the SP and is in charge for initiating the login procedure to the IdP. This is called a **SP-Initiated Login**.

With respect to the Resource Server, **Kong microgateway will be representing the Service Provider (SP)**. Kong will entirely manage the authentication flow, interacting with the end-user and the Identity Provider, validating the SAML assertions and documents exchanged, and more.

The SAML plugin has a lot of configurations available. As stated in the official documentation, the minimum configuration required is:

- The **certificate of the Identity Provider** (`idp_certificate`): the Service Provider (SP) is required to acquire the public certificate from the Identity Provider (IdP) for signature validation. This certificate is kept on the SP and serves to confirm the authenticity of a response originating from the IdP.
- The **Assertion Consumer Path (ACS) Endpoint** (`assertion_consumer_path`): it refers to the endpoint supplied by the Service Provider (SP) where SAML responses are posted. The SP is obligated to furnish this information to the Identity Provider (IdP).
- The **Identity Provider Sign-in URL** (`idp_sso_url`): it denotes the Identity Provider (IdP) endpoint where SAML will initiate `POST` requests. The Service Provider (SP) must retrieve this information from the IdP.
- The **Issuer** (`issuer`): it is the unique identifier of the Identity Provider (IdP) application, specifically the Keycloak client identifier (`client_id`) in this instance.

The first thing to configure is an **anonymous consumer**. If not set, a Kong Consumer must exist for the SAML IdP user credentials, mapping the username format to the Kong Consumer username. With that in mind, the final configuration is the following.

```

1 plugins:
2   - name: saml
3     route: moviescatalog-service_retrieveallmovies
4     enabled: true
5     config:
6       anonymous: anonymous
7       assertion_consumer_path: "/movies"
8       idp_sso_url:
9         ↪ http://keycloak.microgateway:8080/realms/moviescatalog/protocol/saml
10      validate_assertion_signature: true
11      session_secret: <secret>
12      issuer: kong-sp
13      idp_certificate: <PEM_certificate>

```

In particular:

- The `route` parameter indicates which Kong Route the plugin should be applied to. Note that the plugin can also be applied globally or to a broader Kong Service.
- The `anonymous` parameter refers to the anonymous consumer previously mentioned.
- The `assertion_consumer_path` parameter refers to the ACS previously mentioned.
- The `idp_sso_url` parameter indicates where the IdP exposes the sign-in page. Consumers of the protected API will be redirected here.
- The `validate_assertion_signature` enables signature validation for SAML responses.
- The `session_secret` parameter is used as the secret key for encrypting session data as well as state information that is sent to the IdP in the authentication exchange.
- The `issuer` refers to the `client_id` in the Keycloak realm, as previously mentioned.
- The `idp_certificate` is the public certificate provided by the IdP. This is used to validate responses from the IdP.

4.4.3 Running the scenarios

Once the Kubernetes cluster has been correctly configured, the Identity and Access management infrastructure is set up and the Kong microgateway plugins have been installed, it is possible to simulate and understand the authentication and authorization scenarios. As already mentioned, **the API Microgateway Pattern** focuses on the decentralization and delegation of supporting functionalities. Therefore, the presented authentication and authorization flows have been adapted to this specific paradigm.

More specifically, the following list comprehends the five different scenarios that are reported in this study (**each scenario adopts the correspondent terminology defined in the RFCs**):

- **OAuth 2.0 - Authorization Code Flow.** This scenario involves the usage of Keycloak as the Authorization Server (AS), OPA as the Policy Enforcement Point and Kong

microgateway enhanced with the JWT plugin and the OPA plugin.

- **OAuth 2.0 - Client Credentials Flow.** This scenario involves the usage of Keycloak as the Authorization Server (AS), OPA as the Policy Enforcement Point and Kong microgateway enhanced with the JWT plugin and the OPA plugin.
- **OAuth 2.0 - Resource Owner Password Flow.** This scenario involves the usage of Keycloak as the Authorization Server (AS), OPA as the Policy Enforcement Point and Kong microgateway enhanced with the JWT plugin and the OPA plugin.
- **OpenID Connect - Authorization Code Flow.** This scenario involves the usage of Keycloak as the Identity Provider (IdP), OPA as the Policy Enforcement Point and Kong microgateway enhanced with the OIDC plugin and the OPA plugin.
- **SAML 2.0 - Service Provider (SP) Initiated Login.** This scenario involves the usage of Keycloak as the Identity Provider (IdP) and Kong microgateway enhanced with the SAML plugin.

4.4.3.1 OAuth 2.0 - Authorization Code Flow

The **Authorization Code Flow** is an OAuth 2.0 authentication and authorization flow involving the exchange of an **Authorization Code for an Access Token**. It is defined in OAuth 2.0 RFC 6749, section 4.1 [Har12].

This flow is exclusive to confidential applications, like Regular Web Applications, as the authentication credentials of the application are part of the exchange and require secure handling. For these reasons, typical applications in which the Authorization Code Flow is involved, separate a **Web App front-channel** (running insecurely on the browser) from a **Web App back-channel** (running on the backend server, considered secure).

The scenario presented involves the following entities (adapted to the OAuth 2.0 terminology):

- The **Resource Owner** is the end-user that has to grant Clients access to the Resource Server, in this case, the *MoviesCatalog* application.
- The **Client** is the third-party application which is requiring access to the Resource Server on behalf of the user. It is divided in a **front-channel**, running insecurely in the user's browser, and a **back-channel**, running on servers and considered secure enough to hold the Access Token.
- The **Authorization Server** is Keycloak. For simplicity, as Keycloak is deployed only internally the organization cluster and no ingresses are defined in the K8s cluster, a port-forward allows access for external Clients to test the scenario. In a production-ready environment, Authorization Server's endpoints are supposed to sit behind behind a WAF or API Gateway.
- The **Resource Server** is the *MoviesCatalog* application, which exposes the endpoint to add a new movie to the catalog (`POST /movies`). Consumers of this API are required to provide an Access Token. The Resource Server sits behind the **Kong microgateway** instance, that validates the Access Token and authorize the request with **OPA**.

Figure 4.12 lists the steps involved in this authentication and authorization flow.

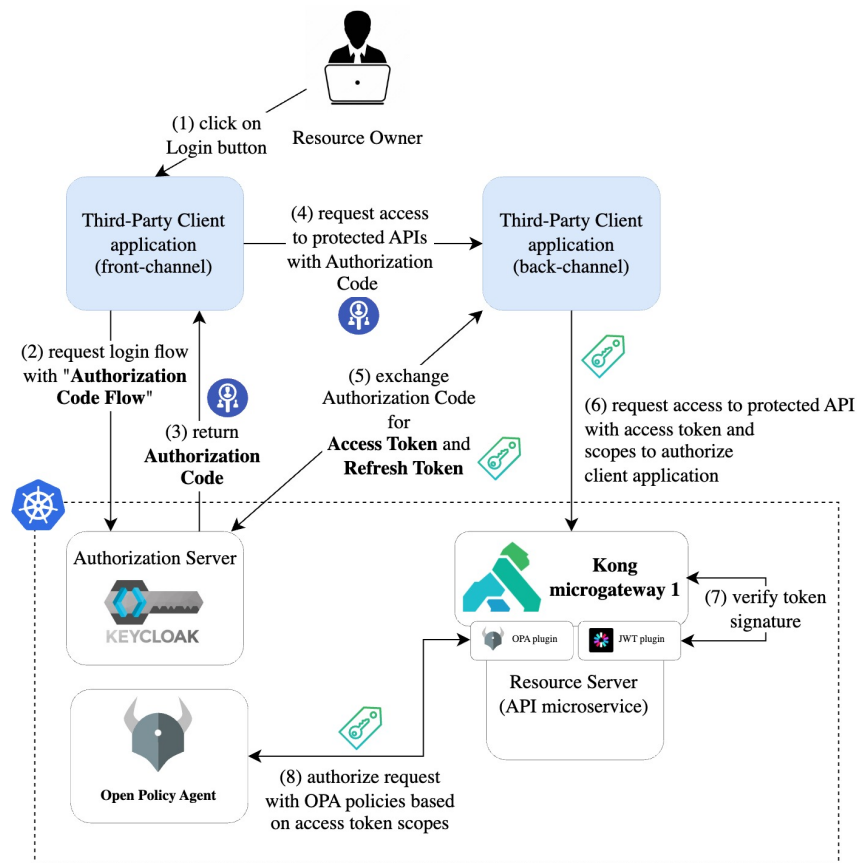


Figure 4.12: Scenario implementing the OAuth 2.0 Authorization Code Flow in Kubernetes with Kong microgateway.

More in details:

1. The **Resource Owner** (the end-user) opens the third-party Web Application (**Client**) on a browser and attempts to access the protected resource without being logged in. This triggers the initiation of the **Authorization Code Flow**.
2. The **Client front-channel** performs the requested login with Authorization Code Flow, redirecting the end-user to the Authorization Server login page. Here, the **Resource Owner** provides the credentials to grant Client access to the Resource Server.
3. The Authorization Server returns the **Authorization Code** that will be later exchanged for an Access Token.
4. The **Resource Owner**, through the Web Application front-channel, requests access to the protected API. To do so, the Client front-channel forwards the Authorization Code obtained in the previous step.
5. The **Client back-channel**, before reaching the protected API, exchange the Authorization Code for the **Access Token** (and **Refresh Token**, to avoid the login from the end-user each time the Access Token expires) as seen in figure 4.13.

- The **Authorization Server** is Keycloak. For simplicity, as Keycloak is deployed only internally the organization cluster and no ingresses are defined in the K8s cluster, a port-forward allows access for external Clients to test the scenario. In a production-ready environment, Authorization Server's endpoints are supposed to sit behind behind a WAF or API Gateway.
- The **Resource Server** is the *MoviesCatalog* application, which exposes the endpoint to add a new movie to the catalog (`POST /movies`). Consumers of this API are required to provide an Access Token. The Resource Server sits behind the **Kong microgateway** instance, that validates the Access Token and authorize the request with **OPA**.

Figure 4.14 lists the steps involved in this authentication and authorization flow.

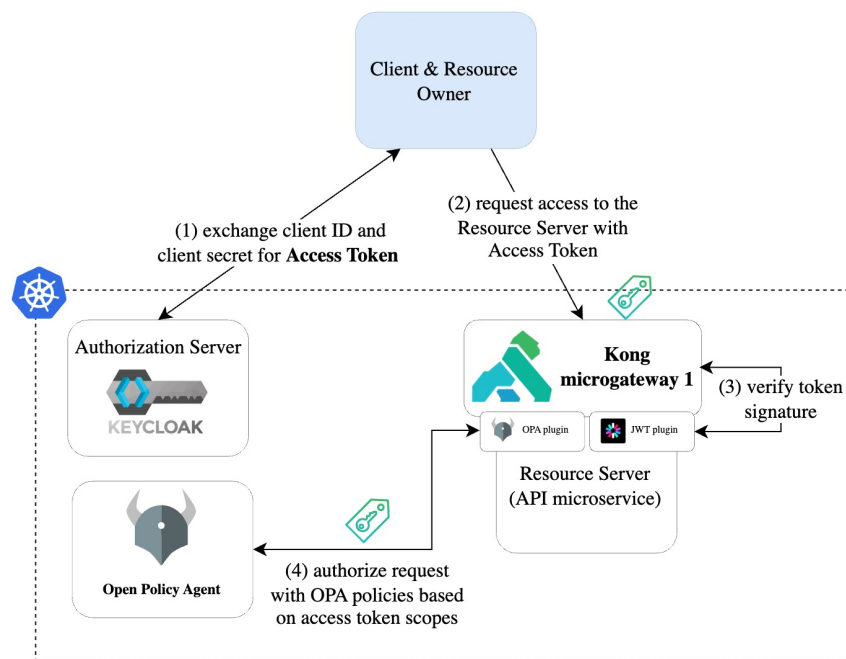


Figure 4.14: Scenario implementing the OAuth 2.0 Client Credentials Flow in Kubernetes with Kong microgateway.

More in details:

1. Client (Insomnia) **exchanges client ID and client secret** with the Authorization Server (Keycloak) for a **signed Access Token (JWS)**.
2. Client makes the request to the protected resource, **forwarding the Access Token**.
3. Kong microgateway **verifys the signature of the Access Token** using the JWT plugin for Kong.
4. Kong microgateway **authorizes the request with fine-grained policies in OPA** using the OPA plugin for Kong.
5. Eventually, the Client **accesses the API** and related resources as show in figure 4.15

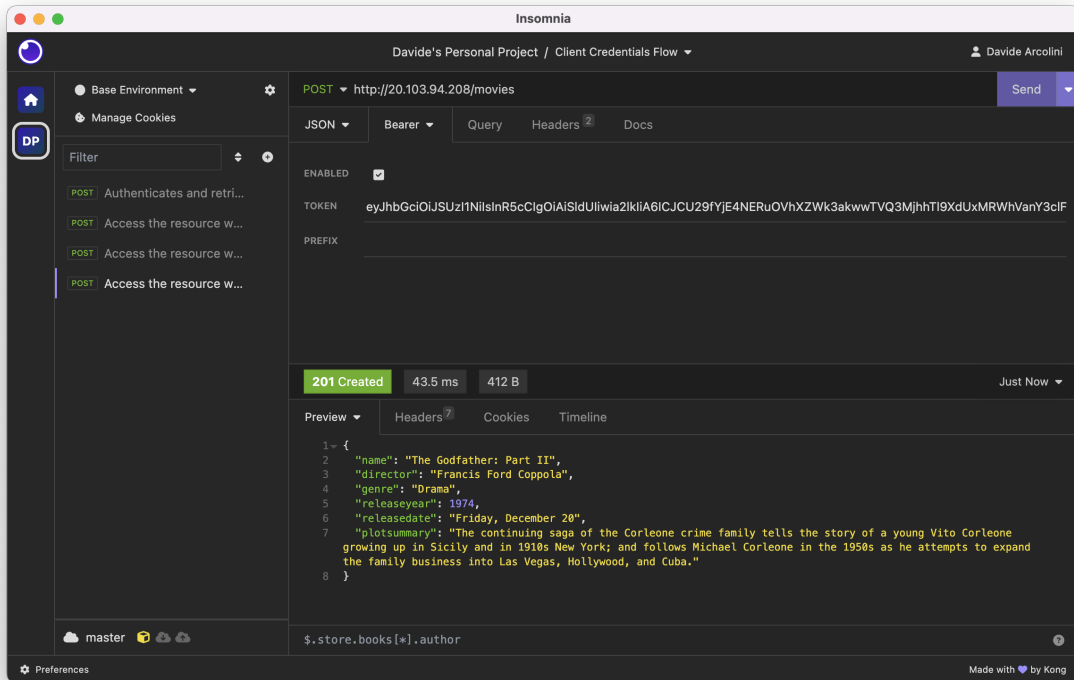


Figure 4.15: Insomnia desktop client simulating the access to the API with the Access Token.

4.4.3.3 OAuth 2.0 - Resource Owner Password Flow

The **Resource Owner Password Flow** is an OAuth 2.0 authentication and authorization flow involving the exchange, typically by using an interactive form, of the **user provided credentials**, such as username and password **for an Access Token**. It is defined in OAuth 2.0 RFC 6749, section 4.3 [Har12].

This flow can only be used for **highly-trusted applications** (such as Organization Applications) because the application's authentication credentials are exchanged and must be kept private. Even if this condition is met, the Resource Owner Password Flow should only be used when redirect-based flows (like the Authorization Code Flow) cannot be used. The scenario presented involves the following entities (adapted to the OAuth 2.0 terminology):

- The **Resource Owner** is the end-user that has to grant Clients access to the Resource Server, in this case, the *MoviesCatalog* application.
- **Client** is the trusted application which is requiring access to the Resource Server on behalf of the user.
- The **Authorization Server** is Keycloak. For simplicity, as Keycloak is deployed only internally the organization cluster and no ingresses are defined in the K8s cluster, a port-forward allows access for external Clients to test the scenario. In a production-ready environment, Authorization Server's endpoints are supposed to sit behind behind a WAF or API Gateway.
- The **Resource Server** is the *MoviesCatalog* application, which exposes the endpoint to

add a new movie to the catalog (`POST /movies`). Consumers of this API are required to provide an Access Token. The Resource Server sits behind the **Kong microgateway** instance, that validates the Access Token and authorize the request with **OPA**.

Figure 4.16 lists the steps involved in this authentication and authorization flow.

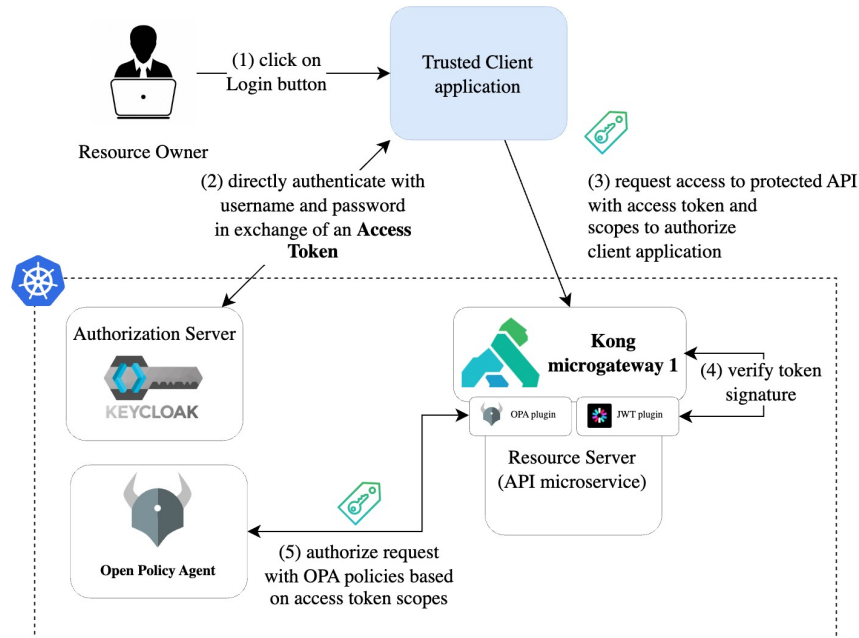


Figure 4.16: Scenario implementing the OAuth 2.0 Resource Owner Password Flow in Kubernetes with Kong microgateway.

More in details:

1. The **Resource Owner** (the end-user) opens a trusted application (the **Client**) and click the login button, providing its own credentials.
2. The **Client**, before reaching the protected API, exchange the end-user credentials for the **Access Token** (and **Refresh Token**, to avoid the login from the end-user each time the Access Token expires).
3. Once the Client possesses the Access Token, it can reach the Resource Server, **forwarding the Access Token**.
4. Kong microgateway **verify the signature of the Access Token** using the JWT plugin for Kong.
5. Kong microgateway **authorize the request with fine-grained policies in OPA** using the OPA plugin for Kong.
6. Eventually, the Client **accesses the API** and related resources.

4.4.3.4 OpenID Connect - Authorization Code Flow

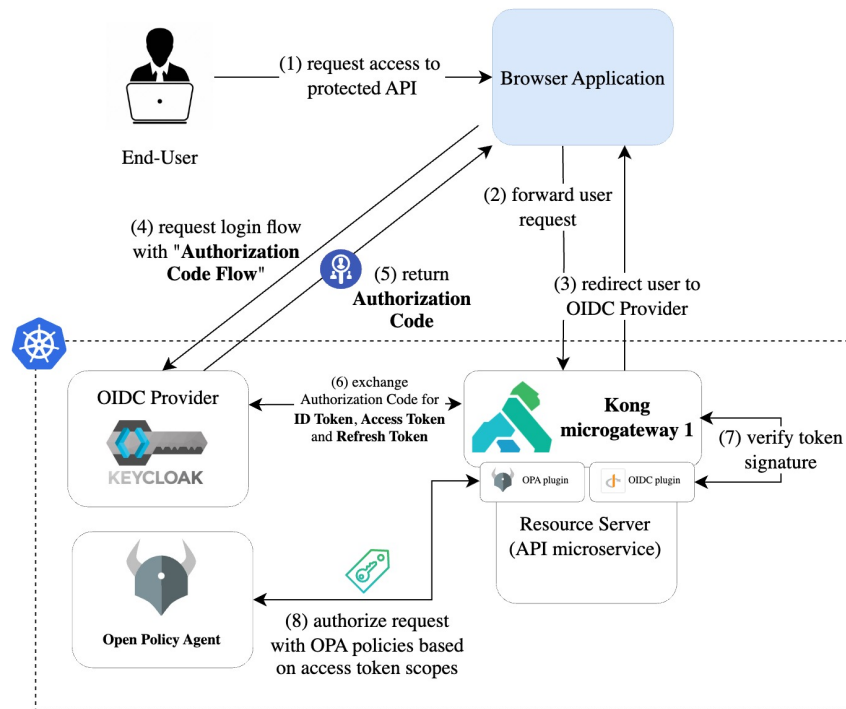


Figure 4.17: Scenario implementing the OpenID Connect Authorization Code Flow in Kubernetes with Kong microgateway.

OpenID Connect is an interoperable authentication protocol built upon the OAuth 2.0 framework. It streamlines the process of verifying user identity through authentication performed by an Authorization Server, facilitating the retrieval of profiles information.

The **Authorization Code Flow with OpenID Connect** is the most versatile and complete Authentication and Authorization flow that can be implemented to grant clients access to protected APIs. This scenario presents again the OAuth 2.0 Authorization Code Flow, adapted to the OpenID Connect standards, implemented with Kong microgateway. Note that OpenID Connect uses slightly different names with respect to OAuth 2.0 Specifications. The scenario presented involves the following entities (adapted to the OpenID Connect terminology):

- **End-User** is the Resource Owner in the OAuth 2.0 domain. It has to grant Clients access to the Resource Server, in this case, the *MoviesCatalog* application.
- **Relying Party** is the Client in the OAuth 2.0 domain. It requests access to the Resource Server on behalf of the user.
- **OpenID Connect Provider** is the Authorization Server in the OAuth 2.0 domain. It issues authorization codes, Access Tokens and ID Tokens to authorized Relying Parties.
- The **Resource Server** is the *MoviesCatalog* application, which exposes the endpoint to add a new movie to the catalog (`POST /movies`). Consumers of this API are required to provide an Access Token. The Resource Server sits behind the **Kong microgateway** instance, that validates the Access Token and authorize the request with **OPA**.

Figures 4.17 and 4.18 list the steps involved in this authentication and authorization flow.

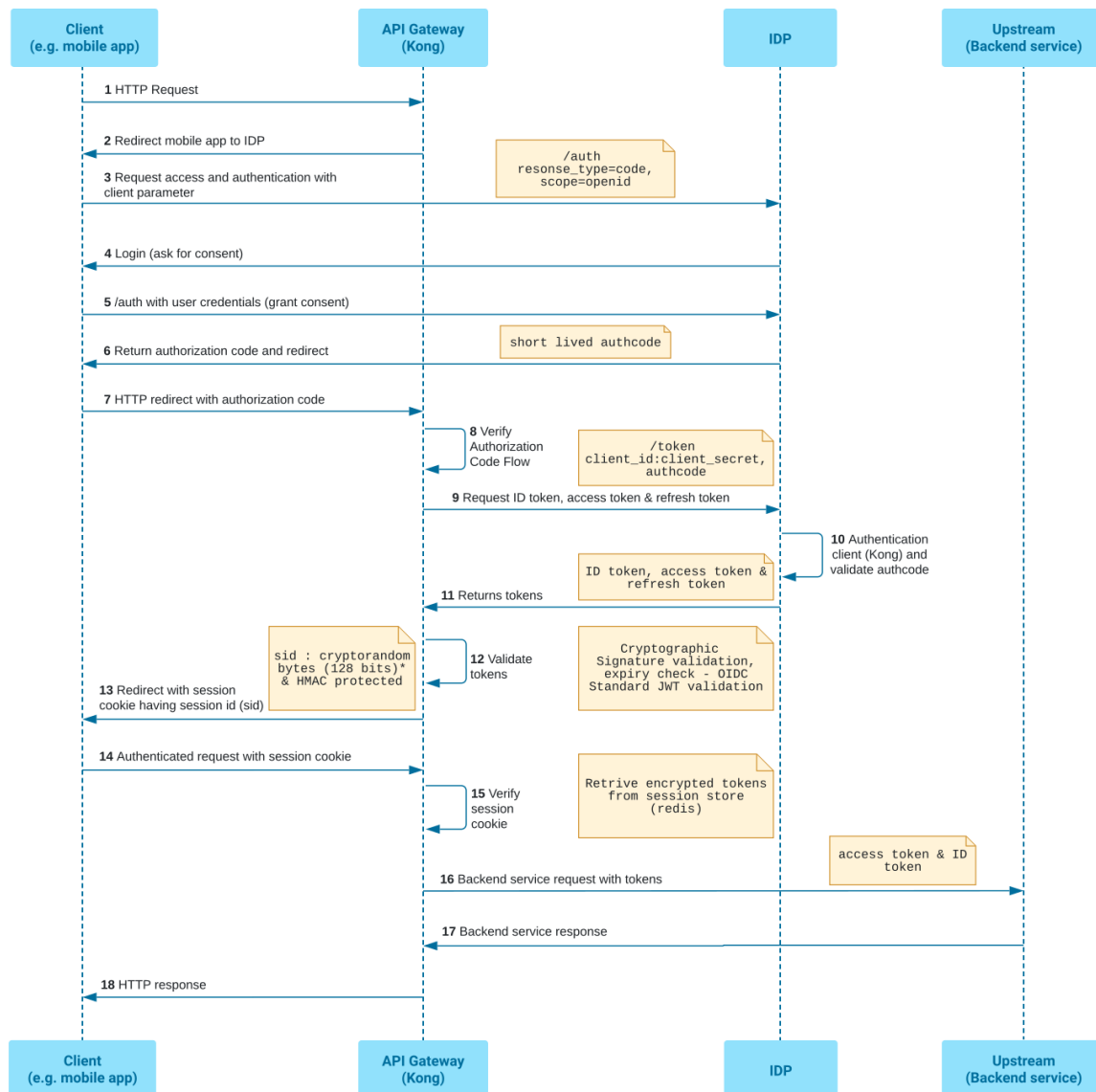


Figure 4.18: Sequence Diagram showing the interaction between the end-user, Kong microgateway and the IdP. OPA integration is missing. [Source: [Kong OpenID Connect](#)]

More in details:

1. **The End-User** (i.e. the Resource Owner) opens the a browser and tries to access the protected API.
2. The **OpenID Connect plugin** for Kong microgateway intercepts the request and redirect the end-user to the OIDC provider, initiating the **Authorization Code Flow**.
3. The end-user performs the IdP login, granting Kong access to the protected API.
4. The OpenID Connect provider (Keycloak) generates an **Authorization Code**, which is returned to the end-user browser with a redirect to the endpoint of the original request.

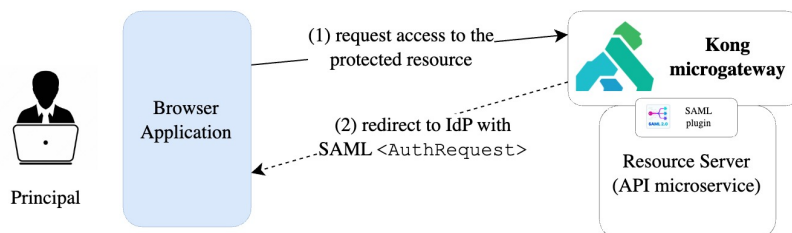
5. The OpenID Connect plugin for Kong microgateway validates the Authorization Code and exchanges it for an **ID Token**, **Access Token** and **Refresh Token**.
6. Kong microgateway **authorize the request with fine-grained policies in OPA** using the OPA plugin for Kong.
7. Eventually, the End-User **accesses the API** and related resources.

4.4.3.5 SAML 2.0 - Service Provider (SP) Initiated Login

Security Assertion Markup Language (SAML), is an **open-standard based on XML** which is used for exchanging authentication and authorization data between parties, particularly in the context of **web-based Single Sign-On (SSO)** and identity management systems. The most current version of SAML is SAML 2.0. SAML enables secure and standardized communication between an **Identity Provider (IdP)**, a **Service Provider (SP)** and the **Principal** (a user). Specifically:

- The **Principal** is almost always a human user who is trying to access a cloud-hosted application.
- The **Identity Provider (IdP)** serves as a cloud-based software solution responsible for storing and verifying user identities, often achieved through a login procedure. Essentially, the IdP's primary function is to assert the individual's identity and their authorized actions.
- The **Service Provider (SP)** is the cloud-hosted application or service the user wants to use. In this case, the SP is exposed through Kong microgateway, which manages the SAML authentication and authorization flow.

As stated in the official documentation [Kon23m], the Kong SAML plugin represents the Service Provider and is responsible for initiating a login with the Identity Provider. This is called an **SP-Initiated Login**. In the SAML 2.0 Service Provider (SP) Initiated Login, the Principal, i.e. the end-user, attempts to access a protected resource: in the considered case, the **POST /movies** API. The SAML plugin for Kong microgateway intercepts the request. The end-user does not have an active logon session and therefore is unauthorized to access the application data.



At this point, the SAML plugin redirects the end-user to the Identity Provider with an authentication request. Specifically, it sends an HTTP redirect response to the browser (HTTP status 302 or 303). The Location HTTP header contains the destination URI of the Sign-On Service at the identity provider together with an **<AuthnRequest>** message, encoded in the

body of the request, named `SAMLRequest`.

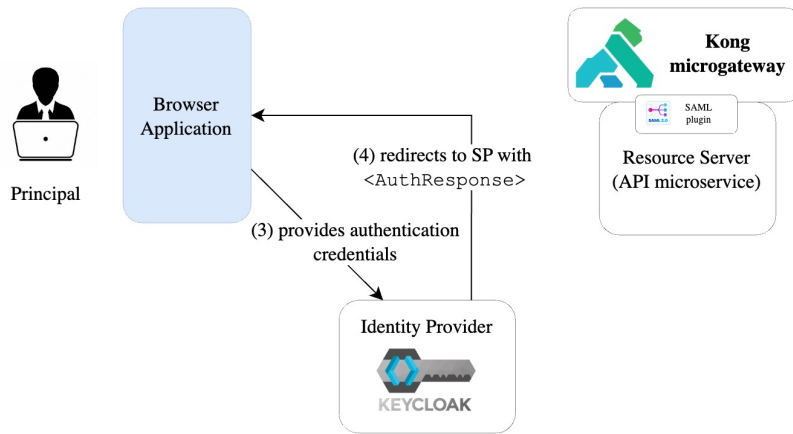
```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <samlp:AuthnRequest
3   xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol"
4   Version="2.0"
5   ID="_c460c4f6-c150-4523-8e7c-ea203ef496eb"
6   IssueInstant="2023-09-27T08:17:10.000Z">
7   <saml:Issuer xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion">
8     kong-sp
9   </saml:Issuer>
10  <samlp:NameIDPolicy
11    AllowCreate="false"
12    Format="urn:oasis:names:tc:SAML:1.1:nameid-format:emailAddress"
13  />
14 </samlp:AuthnRequest>
```

Some important considerations about the SAML request:

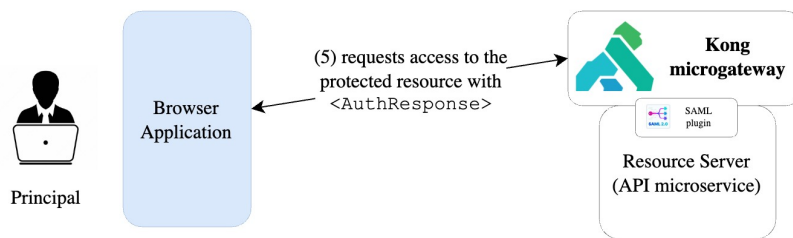
- The `<Issuer>kong-sp</Issuer>` represents the `client_id` in the Keycloak realm. Specifically, from the SAML 2.0 core specifications ([CMJ15]): *"The SAML authority that is making the claim(s) in the assertion. The issuer SHOULD be unambiguous to the intended relying parties"*.
- The `<NameIDPolicy>` outlines restrictions on the name identifier intended to represent the requested subject. In particular, the boolean value `AllowCreate` indicates whether the IdP is permitted, in the process of fulfilling the request, to generate a new identifier that represents the principal.

For this specific use case, the **HTTP Redirect Binding** is used to deliver the SAML `<AuthnRequest>` message to the IdP. The browser processes the redirect response and issues an HTTP GET request to the IdP's Single Sign-On Service with the `SAMLRequest` parameter. The **local state information** (or a reference to it) is also included in the HTTP response encoded in a `RelayState` string parameter.

At this point, the end-user is prompted to post its credentials to the IdP. The IdP Single Sign-On Service builds a SAML assertion representing the user's logon security context. The `<Response>` message is then placed within an HTML FORM as a hidden form control named `SAMLResponse`. If the IdP received a `RelayState` value from the SP, it must return it unmodified to the SP in a hidden form control named `RelayState`. The Single Sign-On Service sends the HTML form back to the browser in the HTTP response. For ease of use purposes, the HTML FORM typically will be accompanied by script code that will automatically post the form to the destination site.



Finally, the SAML plugin for Kong microgateway receives a valid assertion and can grant the End-User access to the protected resource.



5. Enterprise Use-Cases: APIOps with Kong microgateway

The previous chapter presented the details and implementation methodologies that have been developed to adopt the **API Microgateway Pattern** integrated with Kong Gateway. This adoption aligns with industry-recognized best practices in terms of Identity and Access Management (IAM), monitoring capabilities, secrets storage, and more. The scenarios outlined and analyzed represent the concluding phase in a process that encompasses the entire lifecycle of the APIs. This chapter emphasizes all aspects that precede deployment to the Kubernetes cluster, starting from integrating the new source code, automating Kong microgateway configuration, and conducting automated tests for the presented scenarios. Specifically, the focus will be on the API Operations adapted in the CI/CD approach.

As presented in section 2.4.1, **APIOps**, short for API Operations, refers to the set of practices and methodologies aimed at efficiently managing, optimizing, and securing APIs throughout their lifecycle. APIOps applies the concepts of **GitOps** and **DevSecOps** to enable operation team members to easily make iterative and automated changes to API deployment.

"If you cannot efficiently operationalize a technology investment, that investment is wasted. This is no different in the world of APIs and microservices, where every service is designed to support a change to a digital-first culture. APIOps makes this change possible."

- What is APIOps (Kong Blog) [[Hec21](#)]

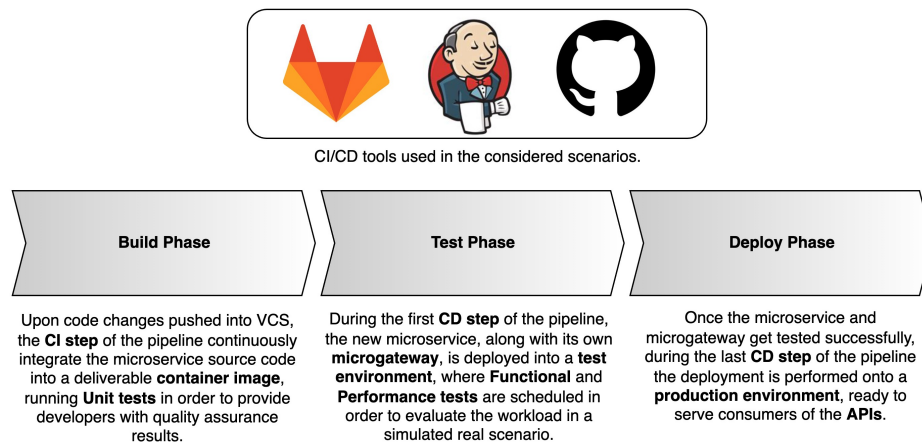


Figure 5.1: The three stages considered in the CI/CD approach developed for this thesis.

The CI/CD pipelines constructed for this research are divided in three main groups. Pipelines start with the **Build Phase**, in which the new API source code is integrated into deliverable Docker container images, running unit tests in order to provide developers with quality assurance results. Then, Kong microgateway is configured and deployed into a test environment along with the microservice it serves. Here functional and performance tests are scheduled in order to evaluate the workload in a simulated real scenario. The last step, the **Deploy Phase**, encompasses the deployment strategies that can be adopted in a real-case scenario (Blue-Green Deployment, Canary Deployment, and more) but they are not considered in this study as they are tailored to the company's policy.

This chapter is divided in three sections:

1. The first section (5.1) focuses on the **infrastructure setup and the tools adopted** in the upcoming workflows. It will present the CI/CD approach used to manage configuration and deployment of the Kong microgateways. The technologies involved in the processes are presented as well.
2. The second section (5.2) focuses on the **Continuous Integration (CI)** part of the pipeline. It will encompass the description of the operations involved to automatically develop, test and integrate APIs with Kong microgateways, using tools like Gradle, 42Crunch or deCK.
3. The third and last section (5.3) focuses on the **Continuous Delivery (CD)** part of the pipeline. In here, deployment capabilities using Kubernetes and testing strategies using Postman/Insomnia and Gatling are presented.

5.1 Automation tools and infrastructure

In the proposed scenarios, although different variations of APIOps will be analyzed, the core idea remains the same: **securely automate the full-lifecycle management of the APIs integrated in the API Microgateway paradigm**. An example of workflow, depicted also in figure 5.2 is the following:

1. Preliminary, the **API Operations team** must have configured the on-premise or remote hosting platform for version control and integration, such as a **self-hosted GitLab** instance or the **on-cloud GitHub** servers. An **automation tool like Jenkins** provides capabilities to run CI/CD pipelines, storing secrets and credentials that are used throughout the execution of the workflow and monitoring the results of each build.
2. The automation starts when the API developers produce **new code for the API** and proceed to open a **Pull Request (PR)** to merge the code into the production branch (or any other target branch that has been configured).
3. When the new code is reviewed and the PR is accepted, merging the two branches, the **CI/CD pipeline is triggered** and the automation server checks out the repository to start working on it.
4. The agent/runner of the pipeline runs **unit and integration tests**, shipping the new image of the microservice exposing the APIs on a on-premise or remote **Docker registry**.

5. The deployment of the new version of the Kong microgateway and microservice happens just right after the two of them are automatically configured, using tools like **Gradle** and **decK**. As soon as the **OpenAPI Specification file** is ready, **42Crunch Security Audits** are performed, blocking the pipeline in case of failure.
6. Finally, once the deployment on the **Kubernetes test environment** is completed, the agent/runner of the CI/CD pipeline triggers the execution of **functional and performance tests**, in order to evaluate the workload in a simulated real scenario.

The main target of APIOps is to securely and fast deliver new APIs to consumers. In the proposed scenarios, this is accomplished with **early-stages analysis of the code and the APIs configurations**, catching and resolving deviations from API standards faster to improve specifications and API quality. As already mentioned, **42Crunch Security Audits** are performed as soon as the OpenAPI Specification file is automatically generated during the build of the application.

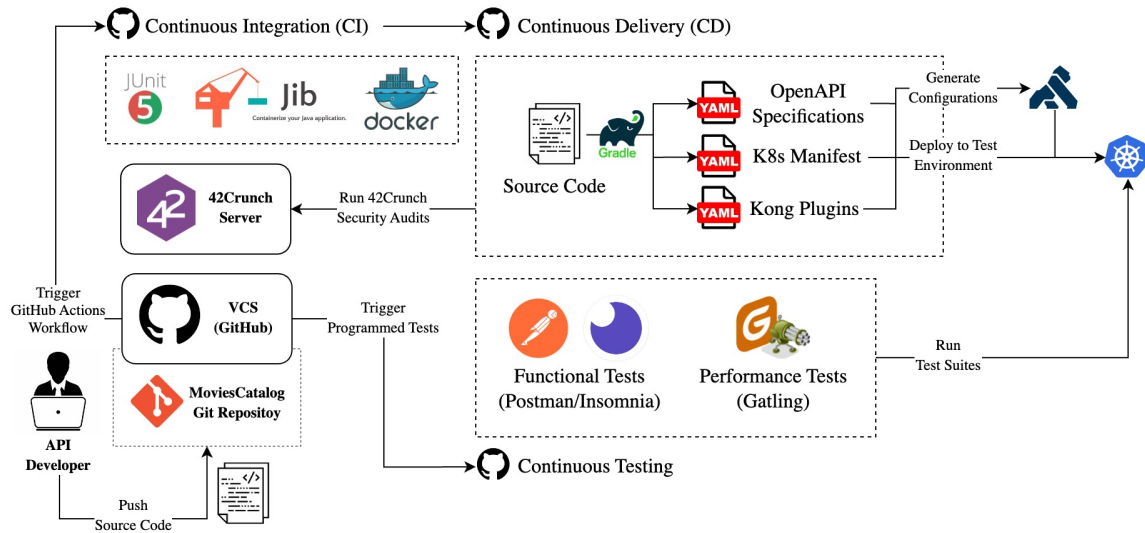


Figure 5.2: API Operations infrastructure and processes in CI/CD pipelines.

This section offers a comprehensive catalog of the various technologies that have been strategically employed in the context of the proposed API Operations. It serves as a detailed reference, encapsulating the wide array of tools, platforms, and solutions harnessed to enhance and streamline the API operations within this framework.

5.1.1 Infrastructure technologies

- **Jenkins**. In the context of this thesis project, the decision to implement Jenkins as the **Continuous Integration and Continuous Delivery (CI/CD) tool** is grounded in its robust capabilities and widespread adoption within the software development community. Jenkins stands out as a reliable and versatile automation server that streamlines the software development pipeline, from code integration to deployment. Its popularity is not coincidental, as it offers several compelling advantages, including **extensive plugin support**, which allows to tailor Jenkins to the specific project requirements. In particular, a **tailored version of Jenkins** has been successfully deployed within

the cluster infrastructure. This implementation was necessitated by the specific requirements of executing custom tools during the CI/CD processes. These tools encompass a range of essential utilities, including Newman, Inso CLI, Docker, Kubectl, and several others, all integral to the seamless and efficient orchestration of the pipeline.

```
1 FROM jenkins/jenkins:latest
2 USER root
3
4 # snippet extracted from the original used Dockerfile
5 RUN apt-get -y install ca-certificates curl gnupg xz-utils
6 [...]
7 RUN apt-get -y install docker-ce docker-ce-cli containerd.io
8   ↪ docker-buildx-plugin
9 [...]
10 RUN apt-get -y install insomnia
11 [...]
12 RUN npm install -g newman
```

- **GitHub** and **GitLab**. GitHub and GitLab are web-based platforms catering to **version control and collaborative software development**. Efficient automation of code production hinges upon a capable hosting environment for sharing API source code among developers and facilitating integration with automation servers for CI/CD pipeline execution. GitHub has been selected for the testing of **GitHub Actions workflows**, effectively eliminating the necessity for self-hosted Jenkins servers. In contrast, GitLab has been deployed as a self-hosted server within the Kubernetes cluster, **seamlessly interfacing with Jenkins to orchestrate CI/CD pipelines**, ensuring robust and streamlined automation capabilities. This strategic selection of platforms optimizes the development and delivery processes, enhancing collaboration and efficiency in software projects.
- **Kubernetes**. Kubernetes, an open-source platform designed for automating deployment, scaling, and containerized application management, stands as the inherent choice when crafting the microgateway architecture. It offers a seamless means of declaratively managing both microservices and Kong instances, effectively integrating into CI/CD workflows. As already mentioned, in this thesis, two Kubernetes clusters have been employed: an on-premise cluster utilizing **microk8s** and a cloud-based **Azure Kubernetes Service** (AKS). These Kubernetes clusters serve as the focal point for all automation procedures, embracing Infrastructure as Code (IaC) principles to efficiently oversee deployments and construct resources consistently, build after build. This strategic selection of Kubernetes environments optimizes the orchestration of microgateway components and paves the way for efficient, scalable, and declarative management within the CI/CD pipeline.

5.1.2 Automation tools

- **Gradle**. In the development of this thesis project, Gradle has been chosen as the build automation and project management tool for several compelling reasons. At first, Gradle is a versatile and powerful build system that simplifies and streamlines the build process, making it well-suited for managing the complexity of this project. Unlike traditional build tools, Gradle uses a **Groovy-based DSL** (Domain-Specific Language) or **Kotlin**,

which offers a more expressive and concise way to define build configurations. This approach enhances readability and maintainability. Gradle's integration with popular IDEs, such as IntelliJ IDEA, enhances the development workflow. While other build tools like Apache Maven were considered, Gradle's flexibility, modern approach, and active community made it the preferred choice for this thesis project.

- **deckK**. `deckK`, standing for *declarative Kong*, streamlines Kong configuration management through a declarative approach, empowering developers to define their desired states, encompassing services, routes, plugins, and more. With `deckK`, the **implementation process is automated** [Kon23c], sparing the manual execution of each step, as is common with the Kong Admin API. It boasts several notable features, such as configuration synchronization with running Kong clusters, drift detection for manual changes, configuration backups, the establishment of automation pipelines with APIOps, and decentralized configuration management using tags, enabling efficient collaboration among diverse teams for configuration distribution. Within this thesis, **deckK has been adopted to enable APIOps** [Kon23d] in the early stage of the design phase.
- **42Crunch**. 42Crunch is a comprehensive platform that specializes in API security, enabling organizations to protect their APIs from potential threats and vulnerabilities throughout the API lifecycle. 42Crunch provides **automated security testing tools** that check APIs for common vulnerabilities, enabling early detection and remediation of potential issues. With its seamless integration into CI/CD pipelines, 42Crunch empowers development and security teams to collaborate effectively and maintain security as a top priority from the early stages of API development to production deployment.
- **Newman** from Postman and **InsoCLI** from Insomnia. **Postman** and **Insomnia** are both popular tools for **managing, testing and debugging APIs**. Firstly, both Postman and Insomnia offer user-friendly interfaces that simplify the process of sending HTTP requests to endpoints, inspecting responses, and validating data and they provide advanced features such as collections and workspaces, which enable the organized and structured management of API requests and tests. Furthermore, the **automation and scripting capabilities** inherent in both Postman and Insomnia empower the creation of test suites and workflows. Specifically, Newman and Inso CLI have been used to automate the tests runs in the APIOps pipelines configured.
- **Gatling**. Gatling is an open-source **performance and load testing framework** that comes with excellent support of the HTTP protocol and provides great versatility. As stated in the documentation [23b], Gatling adopts an asynchronous architecture which is particularly effective when the underlying protocol can be implemented in a non-blocking manner, for instance, the HTTP protocol. This architecture enables the implementation of virtual users as messages rather than dedicated threads, resulting in a resource-efficient approach. Consequently, running numerous concurrent virtual users becomes a seamless process without significant resource overhead.

5.2 Continuous Integration (CI)

This section delves into the **Continuous Integration** (CI) processes tailored to align with the APIOps paradigm. This CI segment within the pipeline is thoughtfully divided into three distinct phases to ensure comprehensive and efficient handling of various aspects:

- **Unit and Integration Phase.** In this initial phase, changes made to the APIs and subsequently pushed to the remote Git repository undergo rigorous testing. JUnit 5 ensures the reliability of these changes. Following successful testing, the updated APIs are integrated into a new Docker image using Google Jib.

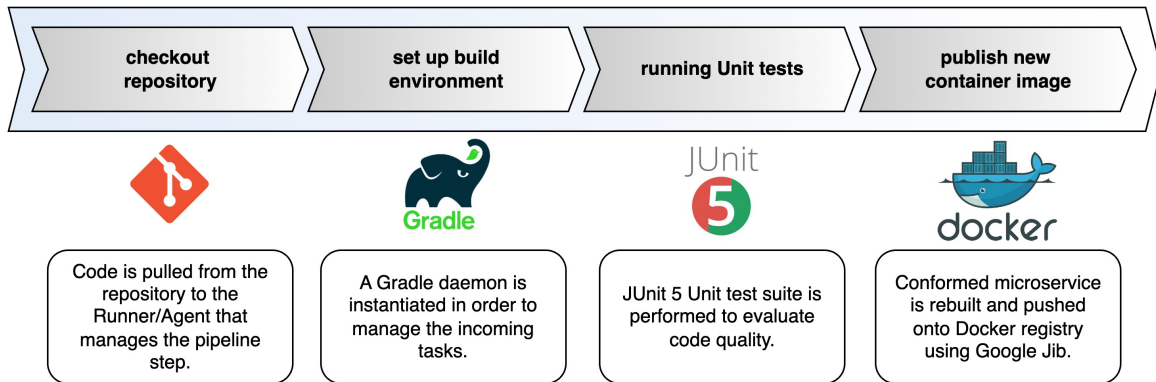


Figure 5.3: Unit and Integration Phase for the CI steps of the pipelines.

- **APIs Management Phase** Moving forward, the focus shifts to the management of the OpenAPI Specification file. This phase involves the generation of the OAS file using Gradle. Additionally, 42Crunch Security Assessments are performed, ensuring that the APIs adhere to the highest standards of security.
- **Kong Microgateway Management Phase.** In the final step of the CI pipeline, attention turns towards the management of the Kong microgateway. Using decK, the OpenAPI Specification file is transformed into a Kong configuration file. This meticulous configuration management guarantees that the Kong microgateway aligns precisely with the API specifications, ensuring robust and reliable service delivery.

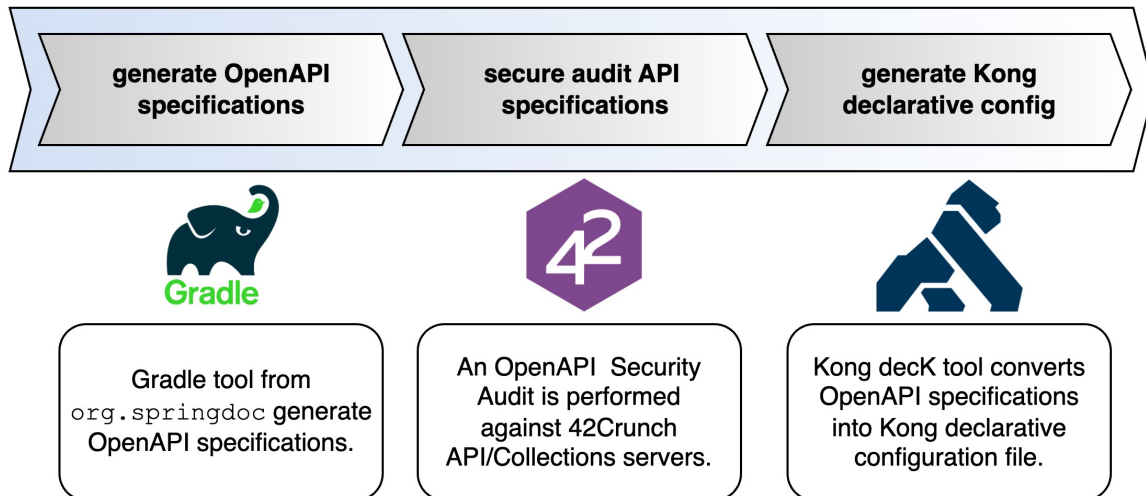


Figure 5.4: APIs and Kong Microgateway Management Phases for the CI steps of the pipelines.

5.2.1 Unit and integration phase

The following subsections present the integration of JUnit and Google Jib in both Jenkins and GitHub Actions.

5.2.1.1 JUnit 5

As presented in section 4.1.2.1, JUnit [Ken23] is a testing framework that facilitates the creation and execution of Java tests, ensuring efficient and reliable testing for Java applications. Running JUnit tests in pipeline is straightforward, thanks to the adoption of Gradle as building tool. The Gradle wrapper, `gradlew` is shipped within the source code and can run out-of-the-box, starting a Gradle daemon. Specifically, in Jenkins, tests are run and results are published with:

```
1 stage("Build and Unit Tests") {
2     steps {
3         sh "./gradlew test"
4         [...]
5     }
6 }
7 [...]
8 post {
9     always {
10        junit "build/test-results/**/*.*.xml"
11    }
12 }
```

While in GitHub Actions:

```
1 jobs:
2   unit-tests:
3     runs-on: ubuntu-latest
4     steps:
5       [...]
6     - name: Running tests
7       run: gradle test --scan
8     - name: Publish Test Results
9       uses: EnricoMi/publish-unit-test-result-action@v2
10      if: always()
11      with:
12        files: build/test-results/**/*.*.xml
```

5.2.1.2 Google Jib

As presented in section 4.1.2.2, **Jib** streamlines container creation, bypassing the need for a Dockerfile [23d]. Its adoption in this research serves three primary objectives: faster updates for microservices, achieved by separating the application into layers; smoother integration with Gradle for enhanced CI/CD convenience; and embracing a daemon-less approach by simplifying the CLI and enabling image building and pushing directly from Maven or Gradle to a designated registry.

Specifically, Jenkins runs the Jib task by defining the `withCredentials` environment. This environment allows the Jenkins node running the current build to extract a set of credentials from the Jenkins Credentials storage. In this case, username and password are needed to authenticate the node to the remote Docker registry:

```

1 stage("Build and Unit Tests") {
2     steps {
3         [...]
4         withCredentials([usernamePassword(credentialsId:
5             ↪ 'registry_dockerhub', usernameVariable:
6             ↪ 'CONTAINER_REGISTRY_USERNAME', passwordVariable:
7             ↪ 'CONTAINER_REGISTRY_PASSWORD')])) {
8             sh "./gradlew jib --image=dvdr00t/api-moviescatalog
9                 ↪ -Djib.to.tags=$BUILD_NUMBER-reply,latest-reply --quiet"
10        }
11    }
12 }

```

In GitHub Action, the operation is pretty similar, except that the set of credentials are extracted from the GitHub Secrets storage:

```

1 jobs:
2   build:
3     runs-on: ubuntu-latest
4     steps:
5       [...]
6     - name: Set up build number
7       id: build-vars
8       run: |
9         echo "IMAGE_TAG=$GITHUB_RUN_NUMBER" >> $GITHUB_ENV
10    - name: Build and push Docker image
11      env:
12        REGISTRY: dvdr00t/api-moviescatalog
13        CONTAINER_REGISTRY_USERNAME: ${ secrets.CONTAINER_REGISTRY_USERNAME
14          ↪ }
15        CONTAINER_REGISTRY_PASSWORD: ${ secrets.CONTAINER_REGISTRY_PASSWORD
16          ↪ }
17      run: gradle jib --image=dvdr00t/api-moviescatalog -Djib.to.tags=${
18          ↪ env.IMAGE_TAG }},latest --quiet

```

5.2.2 APIs management phase

The following subsections present the integration of Gradle and 42Crunch within Jenkins and GitHub Actions.

5.2.2.1 OpenAPI Specification file

The first step in the management of the APIs is to generate the OpenAPI Specification file that is used as reference throughout the entire workflow. As already mentioned, this document serves two purposes:

- It enables developers to automatically perform a Security Assessment of the APIs with 42Crunch.
- It enabled developers to automatically convert the OAS file generated into a Kong declarative configuration file.

Once again, as Gradle is used to manage the generation of the OpenAPI Specification file, running the task is pretty straightforward, both in Jenkins:

```

1 stage("Generate OpenAPI Specification and run 42Crunch tests") {
2     steps {
3         sh "./gradlew generateOpenApiDocs --quiet"
4         [...]
5     }
6 }

```

and in GitHub Actions:

```

1 jobs:
2   manage-api:
3     runs-on: ubuntu-latest
4     steps:
5     [...]
6     - name: Generate OAS file
7       run: |
8         gradle generateOpenApiDocs --quiet

```

5.2.2.2 Security Audits with 42Crunch

Once the OpenAPI Specification file has been generated with Gradle, APIs are documented in a standard format and ready to be employed in the subsequent processes. As the entire configuration of the Kong microgateway instance that will be deployed on the Kubernetes cluster is based on this document, it is mandatory to assert that the APIs definition is compliant with the security best-standards. The OAS file undergoes a 42Crunch Security Assessment in order to stop the pipeline in case of any inconsistency or bad configuration in accordance with OWASP API Security Top 10 and other best-practices for securing APIs.

Running the security scan of the OpenAPI Specification file from CI/CD requires the integration of the 42Crunch Plugin and the employment of an API Token. The [API Token](#), which has specific access rights (scopes) to access 42Crunch Platform, can be generated from the Platform itself with a paid subscription. The means to obtain such token are not presented here. Once the Token is securely stored either in Jenkins Credentials or GitHub Secrets, it can be used by the agent/node running the pipeline to send the OAS file to 42Crunch servers. As specified in the [official documentation](#), the plugin assesses the quality of OpenAPI files within the project. In cases where the identified APIs fail to align with the criteria set in the plugins or the [security quality gates](#) within the 42Crunch Platform, the plugin marks the build as unsuccessful, preventing the progression of substandard APIs to the testing or production staging phases. This ensures that only APIs meeting the specified standards proceed further in the development pipeline.

The plugins work in two phases:

1. **Discovery**: the plugin checks the project for any `.json`, `.yaml`, and `.yml` files. When it finds an OAS target file, if it is `.yaml` or `.yml`, it is automatically converted to a JSON file. Once the conversion is completed, the discovered APIs gets uploaded in the 42Crunch Platform, in a dedicated API Collection.
2. **Audit**: Security Audit scans the new APIs for their integrity, consistency and security. If the APIs meets the quality-level defined in the criteria, the task or job ends succeeds. The CI/CD pipeline processes the result as defined and then continues to the next task or job.

The plugin can be further configured by adding a `42c-conf.yaml` file in the project, to change the behavior of the integration plugin. It's crucial to note that the failure criteria established in the CI/CD plugin, including the minimum score requirement, operate independently of the acceptance criteria defined in the security quality gates (SQGs). Consequently, a CI/CD build may fail due to non-compliance with the plugin's criteria, the SQG criteria, or a combination of both. The minimum score, referred to as `min-score`, represents the lowest audit score that the examined OpenAPI definitions must attain for the build step to be deemed successful. On the other hand, SQGs introduce additional constraints that must be satisfied by the Security Audits for the overall build to proceed successfully. Therefore, the build may encounter failure if any of these criteria are not met during the evaluation process.

The **42Crunch plugin for Jenkins** can be found in the Jenkins update center as [42Crunch REST API Static Security Testing](#). Installing the plugin create both the credentials type, that can be used to store the 42Crunch API token, and the `audit` action that can be used in the pipeline as:

```
1 stage("Generate OpenAPI Specification and run 42Crunch tests") {
2     steps {
3         [...]
4         audit
5             repositoryName: "${env.GIT_URL}",
6             credentialsId: '42crunch-token-id',
7             minScore: 100,
8             jsonReport: '42c-report.json'
9     }
10 }
```

All the configuration properties can be found in the official documentation. In the proposed case, the following keys have been defined:

- `repositoryName` is always required and defines which is the repository containing the project to scan.
- `credentialsId` retrieves the 42Crunch API token from the secret credentials defined in Jenkins.
- `minScore` sets to 100 the minimum audit score for the build to succeed.
- `jsonReport` sets the name of the report file that will be later used to view the results.

The **42Crunch plugin for GitHub Actions** can be found in the Actions marketplace as **42Crunch REST API Static Security Testing**. The name of the action is `api-security-audit-action` and can be easily integrated in a GitHub actions by adding the following snippet of code to the workflow file:

```
1 jobs:
2   manage-api:
3     runs-on: ubuntu-latest
4     steps:
5       [...]
6     - name: Run 42Crunch API Audits
7       uses: 42Crunch/api-security-audit-action@v3
8       with:
9         api-token: ${{ secrets.API_TOKEN }}
10        min-score: 100
11        json-report: 42c-report.json
```

All the configuration properties can be found in the official documentation. In the proposed case, the following keys have been defined:

- `api-token` retrieves the 42Crunch API token from the secret variables defined in the GitHub workflow.
- `min-score` sets to 100 the minimum audit score for the build to succeed.
- `json-report` sets the name of the report file that will be later used to view the results.

The OpenAPI Spec file is generated by the upstream service using the Gradle automation tool and configurations that have been defined by the developers team. However, in the context of **microgateways**, a lot of features, like security itself, are not defined at the API level. Rather, it is the microgateway itself that handle all the aspects. For these reasons, it may be possible that the generated OpenAPI Specification file does not contain certain properties that are required by 42Crunch to audit the APIs. To overcome such difficulties, it is possible to add the `42x-extensions` to the OAS file that changes the behavior of the audits.

As an example, the `x-42c-accept-empty-security` can be added to allow empty security schemes in the definition of the Security objects in Path requests:

```
1 openapi: 3.0.0
2   [...]
3   x-42c-accept-empty-security: true
4   [...]
5   post:
6     description: Creates a new movie in the catalogue.
7     operationId: createMovie
8     security: []
```

5.2.3 Kong microgateway management phase

After the OpenAPI Specification file is validated with 42Crunch Security Audits, it can be converted to a `kong.yaml` configuration file for the Kong microgateway. To generate

such configuration, the plugins to be installed in Kong are added and the configuration will be loaded at boot-time.

5.2.3.1 Converting OAS to Kong and validating

As decK is already installed in the self-managed Jenkins instance, running and validating the generation of the Kong configuration file is straightforward (see commands in 3.3.3.1):

```
1 stage("Generate Kong configuration and API tests") {
2     steps {
3         sh "deck file openapi2kong --spec build/openapi.yaml --format yaml
4         ↪ --select-tag moviescatalog-team --output-file kong.yaml"
5         sh "deck file merge kong.yaml kong-plugins.yaml kong-vaults.yaml
6         ↪ kong-consumers.yaml --output-file kong.yaml"
7         sh "deck validate --state kong.yaml"
8     }
9 }
```

On the other hand, in GitHub Actions the tool has been installed manually to provide more control over the version used with respect to the community-developed actions. Once installed, the steps are the same.

```
1 jobs:
2   manage-api:
3     runs-on: ubuntu-latest
4     steps:
5     [...]
6     - name: Install decK
7       run: |
8         curl -sL https://github.com/kong/deck/releases/download/v1.25.0/...
9         ↪ -o deck.tar.gz
10        tar -xf deck.tar.gz -C /tmp
11        sudo cp /tmp/deck /usr/local/bin/
12    - name: Generate Kong configurations
13      run: |
14        sh "deck file openapi2kong --spec build/openapi.yaml --format yaml
15        ↪ --select-tag moviescatalog-team --output-file kong.yaml"
16        sh "deck file merge kong.yaml kong-plugins.yaml kong-vaults.yaml
17        ↪ kong-consumers.yaml --output-file kong.yaml"
18        sh "deck validate --state kong.yaml"
```

5.3 Continuous Delivery (CD)

This section delves into the **Continuous Integration** (CI) processes tailored to align with the APIOps paradigm. This CI segment within the pipeline is thoughtfully divided into three distinct phases to ensure comprehensive and efficient handling of various aspects:

- **Deployment Phase.** The deployment phase is straightforward: involving principles of Infrastructure as Code (IaC), the state of the Kubernetes test cluster is kept into

a Kubernetes Manifest file called `kubernetes-deployment.yaml`. This file contains the configurations related to the microservice deployment, so that when the image of the APIs change, the deployment can be rolled out, the new image can be installed in the container and the pods can be re-instantiated on the cluster.

- **Test Phase.** The test phase is the final gatekeepers that ensure an API's reliability and performance in real-world scenarios. **Functional tests** validate that the API behaves as intended, preserving data integrity and preventing unexpected errors, while **stress tests** push it to its limits to ensure it can withstand heavy loads and maintain responsiveness.

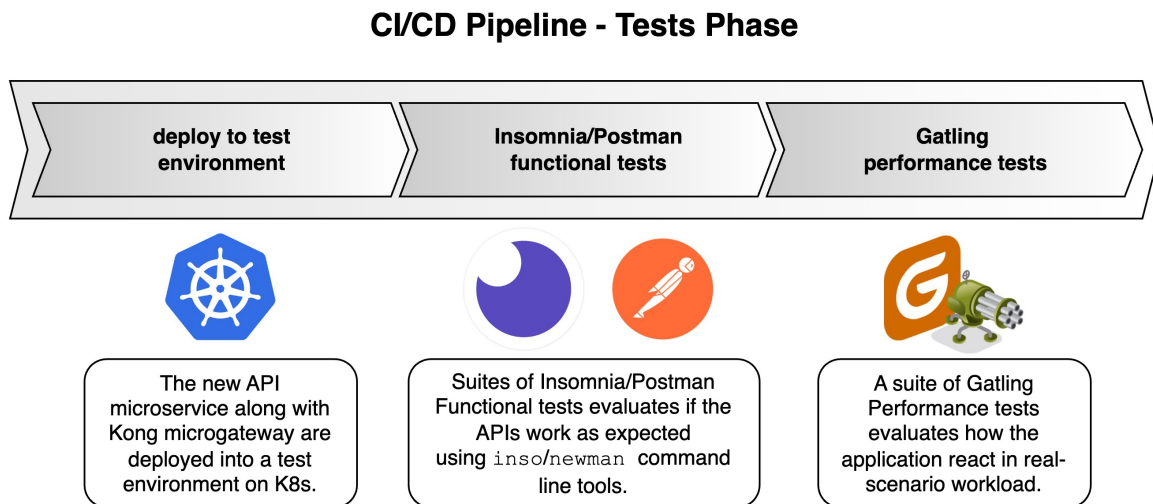


Figure 5.5: Deployment and Test Phase for the CD steps of the pipelines.

5.3.1 Deployment phase

The following subsections present the integration of Kubernetes operations in both Jenkins and GitHub Actions.

5.3.1.1 Rolling-Out Kong microgateways

Before starting the rollout of the deployment in the test environment, it is necessary to configure correctly the `kubectl` tool which will be used by the agent running the pipeline. Therefore, it is mandatory to indicate where the Kubernetes cluster can be found and which are the credentials to access it.

In Jenkins, it is possible to configure a **Cloud environment** by specifying where the K8s cluster is located, at which port Jenkins can connect to it and which is the `config` file that can be used to configure the `kubectl` command. More specifically:

1. It is necessary to install the **Kubernetes plugin for Jenkins**. This will introduce new type of credentials, called **Kubernetes Service Account** and **Kubernetes configuration (kubeconfig)**.
2. Configure the credentials by providing a `kubeconfig` file or a dedicated service account for Jenkins.

3. Create a new Cloud Environment and define the location of the cluster and the credentials to access it.

Once all the steps are completed, it is possible to run `kubectl` commands from the pipeline as:

```
1 stage("Deploy to Test Environment") {
2   steps {
3     sh "kubectl create configmap kong-config-moviescatalog
4     ↪ --from-file=kong.yaml --dry-run=client -o yaml | kubectl apply -n
5     ↪ microgateway -f -"
6     sh "kubectl apply -f kubernetes-microgateway.yaml"
  }
}
```

In GitHub Actions, the process of configuring `kubectl` is even more straightforward. There are several actions already built to configure the tool. As an example, it is possible to:

1. Create a new Secret variable called `KUBE_CONFIG`.
2. Add the `setup-kubectl` actions from [ThomasKliszowski](#):

```
1 jobs:
2   deploy-kong:
3     runs-on: ubuntu-latest
4     steps:
5     [...]
6     - name: Configure Kubectl
7       uses: ThomasKliszowski/setup-kubectl@v1
8       with:
9         kube-config: ${{ secrets.KUBE_CONFIG }}
10        kube-version: 1.27.2
```

3. Directly run the command inside the job:

```
1 jobs:
2   deploy-kong:
3     runs-on: ubuntu-latest
4     steps:
5     [...]
6     - name: Deploy to K8s
7       run: |
8         kubectl create configmap kong-config-products
9         ↪ --from-file=kong.yaml --dry-run=client -o yaml | kubectl
10        ↪ apply -n microgateway -f -
11        kubectl apply -f kubernetes-microgateway.yaml
```

5.3.2 Test phase

The following subsections present the integration of Postman, Insomnia and Gatling tests in both Jenkins and GitHub Actions.

5.3.2.1 Functional tests with Insomnia

[Insomnia](#) is an open-source desktop application for API design, debugging, and testing. While it's valuable for manual API requests and response evaluation, in CI/CD pipelines, a more suitable tool is [InsoCLI](#). It leverages Insomnia's core libraries and Node.js, making it ideal for using Insomnia's functionalities in terminal applications and CI/CD environments.

Using Inso CLI, functional test collections can be seamlessly integrated into Jenkins pipelines or GitHub Actions workflows. By creating and exporting collections in Insomnia and pushing them to the Git repository, pipeline agents can utilize them to perform HTTP requests and validate responses. In this scenario, Inso CLI version 3.18.0 is employed. It's directly installed within the Jenkins server as previously outlined, while in GitHub Actions workflows, Inso CLI is dynamically installed during job execution.

```
1 jobs:
2   functional-tests:
3     runs-on: ubuntu-latest
4     steps:
5     - name: Set up Inso CLI
6       uses: kong/setup-inso@v1
7       with:
8         inso-version: 3.18.0
```

The test suite is created inside the Insomnia desktop application and exported as YAML file once ready to be used inside the CI/CD pipeline for the deployment. Although Cloud synchronization capabilities exist, they require API Tokens to run, losing, in addition, the possibility to keep the file under the same Version Control System of the application.

Both in Jenkins and GitHub Actions, to run the collection with Inso CLI, the following command is used:

```
1 inso run test MoviesCatalog \
2   --src ./tests/MoviesCatalog-FunctionalTests.insomnia_collection.yaml
```

5.3.2.2 Functional tests with Postman

While using Kong Insomnia to run functional tests over Kong microgateways is the most natural answer, the choice of using Newman rather than Inso CLI comes from the lack of possibility for issuing reports in the pipeline with Insomnia.

Newman is a command-line collection runner for Postman. With Newman, it is possible to run a functional tests collection directly in the Jenkins pipelines or GitHub Actions workflows. By creating and exporting collections in Postman, those can be pushed to the git repository in such a way that the Agent/Runner of the pipeline can use them to make HTTP requests and validate the responses accordingly.

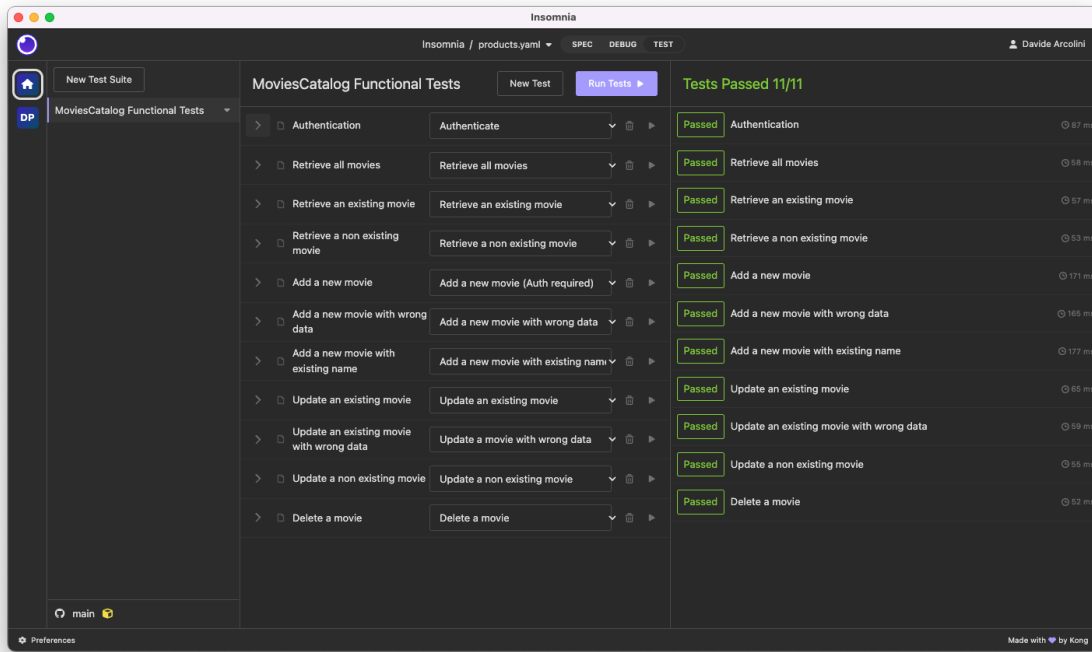


Figure 5.6: Test suite containing functional tests designed in Insomnia.

In the proposed scenario, Newman version `5.3.2` has been used. Specifically, Newman has been directly installed inside the Jenkins server, as already mentioned. As for the GitHub Actions workflows, Newman has been installed dynamically during the execution of the job:

```

1 jobs:
2   functional-tests:
3     runs-on: ubuntu-latest
4     needs: deploy-to-test
5     steps:
6     [...]
7     - name: Install newman
8       run: |
9         npm install -g newman
10        npm install -g newman-reporter-htmlextra

```

As for the Insomnia test suites, the same workflow applies to the Postman collection. In that case, collections are exported as JSON file, where items represent the tested HTTP requests. To run the collection with Newman, the agent running the pipeline (both in Jenkins and GitHub Actions) launch the following command in pipeline:

```

1 newman run ./tests/MoviesCatalog-FunctionalTests.postman_collection.json \
2   -r htmlextra \
3   --reporter-htmlextra-export testResults/htmlreport.html

```

5.3.2.3 Performance tests with Gatling

[Gatling](#) is a powerful open-source load testing framework that allows to simulate high levels

of concurrent users and analyze system's performance under stress. Gatling is "*designed for continuous load testing and integrates with your development pipeline. Gatling includes a web recorder and colourful reports*".

In the testing phase, it's crucial to evaluate the performance of the Kong microgateways on the cluster, ensuring a comprehensive assessment of plugin configurations and their alignment with specified requirements. Section 4.1.2.4 provides specifications on how to configure the Gradle plugin. In order to execute the simulation using Gradle, it's essential to specify the `Simulation()` class that should be run, along with the necessary environment parameters tailored to the specific requirements of the proposed scenario. The following snippet of code runs Gatling simulations in both the Jenkins pipelines and the GitHub Actions workflows prepared:

```
1 gradle gatlingRun \  
2   -Dgatling.simulationClass=moviescatalog.MoviesCatalogSimulation \  
3   -DUSER_COUNT=10 \  
4   -DRAMP_DURATION=20
```


6. Enterprise Use-Cases: Test Result and Architectures Validation

This chapter outlines the procedures employed to **test and validate** the objectives of this research. It presents and analyzes the results obtained in the development of one specific scenario among the whole set of use-cases studied throughout the course of this project, as shown in the previous chapter. It **defines the methodologies applied to validate the proposed solution**, presenting the results to the reader. Because of the scenarios proposed adopt the newly defined **Microgateway Infrastructural Pattern**, implementing authentication and authorization capabilities, monitoring systems, rate-limiting, secrets management and by following the APIOps best-practices for CI/CD pipelines, it is mandatory to validate the architectures in order to be compliant with the industry standards that regulates the secure exposure of Web APIs. Specifically, the rest of this chapter is composed of:

- Section 6.1, "*Introduction*" represents an introduction to the reasons behind the need for this chapter and the various tools used. Once again, the **aims of the thesis** are exposed clearly, serving as the foundation for understanding the subsequent validation processes. A **brief recap of the work done** is presented as well. This recapitulation encapsulates the key milestones, methodologies employed, and the essential stages of the research and development process. This section enables readers to gain a holistic understanding of the research journey, setting the stage for the validation methodologies.
- Section 6.2, "*Test and validation methodologies*", is a comprehensive exposition of the methodologies used to validate the work and the architecture proposed throughout the thesis. This section illustrates the general framework that has been applied to the whole set of different scenarios. Although **a single scenario will be presented** in the next session, the methodology has been applied to all the scenarios developed and results can be found on the GitHub repository hosting the source code and the documentation of the work.
- Section 6.3, "*Presentation of the results*", comprehends a selection of representative results obtained during the course of this research. These results are presented in a clear and organized manner, offering insights into the outcomes of the validation processes. This serves to substantiate the effectiveness and validity of the methodologies employed, as well as the **overall contributions of this thesis to the field of study**.

6.1 Introduction

As this chapter tries to answer the question: "*why, and how, are validation and testing performed?*", this specific section focuses on the "*why*". It presents the **reasons behind the need to validate the proposed solutions** and the tools used to accomplish such objectives. It states the premises and it briefly recaps the work done, in order to set the stage for the next section, which tries to answer the "*how*" part of the question.

When it comes down to present and deliver a new product, it is crucial to assert that every component works functionally, efficiently and safely. This is the case considered here. The "product" is, indeed, the whole process of **API management in a microgateway infrastructural scenario**. There is a multitude of processes involved in this work, and each of them requires to undergo strict tests analysis and validation. As an example, in this thesis, multiple authentication and authorization flows are presented, some using OAuth, others using OpenID Connect or even SAML. Each of the considered scenarios integrate Kong microgateway instances with the AuthN/AuthZ industry standards. It is granted that security is a fundamental aspect when it comes to APIs: for this reason, I had to **validate the proposed solutions** in order to show that a microgateway approach can be integrated respecting the security constraints.

To accomplish such objective, each setup of the architectures has been analyzed with the thesis' tutor supervisor and it has undergone a series of functional tests to showcase that everything works as expected. It will be shown later in this chapter *how* every "component" is actually validated and tested. A comprehensive list of tools and frameworks used to perform the validation can be found in the table below:

Tool adopted	Version	Description
JUnit 5	1.16.3	Run unit and integration tests on each build.
Google Jib	3.1.4	Build and push Docker images to a remote registry on each build.
42Crunch	/	Run API Security Audits on each build.
Jenkins	2.414.1	Automation self-hosted server hosting the CI/CD pipelines.
Kubernetes	1.26.6	Orchestrate the microservices deployment.
deckK	1.25.0	Convert OpenAPI Specification files to Kong configurations.
Postman	5.3.2	Run functional tests for Kong microgateway on each build.
Insomnia	3.18.0	Run functional tests for Kong microgateway on each build.
Gatling	3.9.5	Run performance tests for Kong microgateway on each build.
Grafana	10.1.2	Monitor Kong microgateway performances.

Table 6.1: Table of tools with descriptions

6.1.1 Objectives of the thesis

The aim of this thesis is to realize, in a laboratory environment, a scenario of **full-lifecycle API management** based on the **microgateway architectural model**, using Open-Source and Enterprise versions of **Kong Gateway**. In alignment with industry recognition, as stated

by Gartner [Sha22], Kong stands as the preeminent leader in the API Gateway landscape. For this reason, the aim of this work is to comprehend and analyze diverse requirements for integrating Kong microgateways with the latest APIOps best-practices. Specifically, this chapter analyzes the following objectives:

- Integrate the scenario with **automation tools** to manage the generation of OpenAPI Specifications and the conversion to Kong declarative configuration files.
- Integrate the scenario with **security assessment tools** for securely developing APIs.
- Develop production-ready use cases, based on the **authentication and authorization industry-level standards**, such as OAuth 2.0 and OpenID Connect.
- Integration with **observability and monitoring tools** to evaluate the performances of the presented scenario.
- Integration with tools following **Infrastructure as Code (IaC)** practices to manage the Kubernetes cluster.

However, note that the objectives outlined here are specifically focused on the validation and testing of the thesis. These objectives are a subset of the comprehensive list found in the introductory chapter, which also highlights the broader objectives related to the foundational work that culminated in the creation of this final product.

The entire lifecycle of the APIs microservices is managed under strict observation: from the very early designing phase, developers can have an idea of how well the APIs will perform, if they are secure enough for the industry standards (e.g. OWASP API Security Top 10) and if they match a valid configuration for Kong microgateway. Ad-hoc scenarios are tested with API Collection functional test suites, ensuring the validity of the infrastructure (e.g. unauthenticated consumers can not access the protected APIs). Everything undergoes monitoring with Prometheus and Grafana, so that developers can have the situation under control even in test environments.

6.1.2 Work recap

Before proceeding to the presentation of the test and validation methodologies adopted, a **brief recap of the research work** is proposed, enabling readers to gain a holistic understanding of the research journey and the key milestones reached.

As already stated, the primary objective of this thesis is to study, develop and showcase infrastructure adopting the API Microgateway Pattern employing Kong Gateway open-source (OSS) and Kong Gateway Enterprise. The methodologies and findings documented in this thesis had been elaborated upon in two distinct chapters following the theoretical background introduction, which had covered essential concepts such as full-lifecycle API management, Kong, and Kubernetes. Specifically, the two main chapters had covered:

1. Chapter 4: "Enterprise Use-Cases: Exposing APIs with Kong microgateway"
2. Chapter 5: "Enterprise Use-Cases: APIOps with Kong microgateway"

To accomplish the first objective (1), the initial step involved designing and developing a robust yet comprehensive Gradle-based Spring Boot CRUD application, written in Kotlin:

the *MoviesCatalog* application. This application served as the foundational element, allowing complete control over the APIs, thus facilitating integration with specific Gradle-based tools that played a pivotal role throughout the project. These tools encompassed essential components such as JUnit for testing, OpenAPI Specifications for precise API documentation, Google Jib for streamlined containerisation, and more. The Spring Boot application designed as a lightweight microservice following best practices, intentionally omits certain functionalities such as authentication, authorization, and comprehensive monitoring. This approach aligns with the microservice pattern, where the role of this specific service focuses solely on the management of resources. To address this, the deployment of the microservice alongside instances of Kong microgateway, both the open-source version and Kong Gateway Enterprise, have been evaluated. This exploration involved an array of plugins tailored to suit each unique use-cases considered, spanning various authentication scenarios (OAuth 2.0, OpenID Connect, SAML 2.0) integrating external Identity Providers and Authorization Servers like Keycloak, as well as policy enforcement tools like Open Policy Agent (OPA). Additionally, Prometheus and Grafana are employed to monitor the status of the cluster, while HashiCorp Vault is utilized for secret management.

Next, the focus shifted towards the integration of DevSecOps with Kong (2), an essential step in achieving full-lifecycle API management for the showcased scenarios. Leveraging powerful automation tools such as Jenkins and GitHub Actions, fast and secure CI/CD pipelines have been established. These pipelines efficiently managed the entire software development lifecycle, from source code management to comprehensive testing. They automatically built, tested, and pushed tagged Docker images to a designated Docker registry, rendering them ready for deployment to the Kubernetes cluster. Subsequently, the pipelines orchestrated the generation and validation of OpenAPI Specifications while conducting Security Audits against 42Crunch servers. Configuration of Kong Gateway instances was also automated using deck automation tool, ensuring seamless and consistent deployment across different environments. Upon completion of the automated processes, the solutions were deployed to a dedicated test environment within the Kubernetes cluster. Here, functional tests were performed using tools such as Insomnia and Postman, guaranteeing the correctness of the APIs. Furthermore, load and stress tests were executed using Gatling to assess performances under heavy loads. The entire deployment process was closely monitored by Prometheus servers, continuously scraping data from Kong instances, thereby providing valuable insights presented through Grafana dashboards, ensuring the reliability and efficiency of the integrated solutions.

6.2 Test and validation methodologies

This section will delve into the **methodologies employed to validate the objectives of this thesis**, focusing on testing and ensuring the integrity and effectiveness of the proposed solutions. The validation process encompasses a series of rigorous steps designed to ensure that the developed APIs, microservices, and Kong microgateway instances are functional, secure, efficient, and compliant with industry standards.

This serves to define the foundational techniques used throughout the thesis: it describes the general testing and validation processes that have been employed in all the considered scenario, without going into the specific details.

- Details on a specific scenario (GitHub Actions workflow with 42Crunch Security Audits and OAuth 2.0 Code Credentials Flow) can be found in Section 6.3.

- Details on the other scenarios can be found in Chapter 4 and Chapter 5 of this document.

The methodologies applied are divided into the following sections:

1. **Continuous Integration and Continuous Delivery (CI/CD)**: during the CI/CD executions, multiple tests and validations are performed. JUnit 5 tests are run upon build of the application; OpenAPI Specifications file are generated, tested and audited with 42Crunch Security Audits functionalities; Kong configuration files are validated with decK and the deployment on the K8s cluster is performed.
2. **Functional Tests**: suites of functional tests are executed with Postman and Insomnia, to assert that the proposed microgateway solution integrates well with authentication and authorization industry standards (e.g. OAuth 2.0 or OpenID Connect).
3. **Performance Tests**: scenarios of load and stress tests are executed with Gatling, to assert that the proposed solution is able to undergo situations of high traffic.
4. **Real-Time monitoring**: integration with observability and monitoring tools (Prometheus and Grafana) is tested. Developers can monitor the traffic on the cluster in real-time thanks to the Kong microgateway plugins.

6.2.1 Continuous Integration and Continuous Delivery (CI/CD)

The validation process begins with **unit and integration testing** using **JUnit 5**. These tests are designed to verify that the logic of the source code functions as expected and that the application interacts correctly with persistence entities. For example, tests might evaluate whether the application correctly handles scenarios such as duplicate entries in a database or unauthorized access attempts. JUnit 5 tests are executed as part of the continuous integration process, triggered automatically when changes are pushed to the repository. Test results are published, providing developers with immediate feedback on the integrity of the code.

Once the unit and integration tests pass, the **Google Jib** tool is employed to build, tag, and push the Docker image of the application to a remote Docker registry. This step ensures that the application is correctly packaged and ready for deployment.

Security is a paramount concern when dealing with APIs. To validate the security of the developed APIs, an automated security assessment is performed using **42Crunch**. The OpenAPI Specifications (OAS) file generated from the source code is submitted to 42Crunch for a thorough security audit. The validation process includes checks for security vulnerabilities, compliance with industry security standards, and adherence to best practices. A minimum score threshold is set to ensure that the API meets stringent security requirements.

Configuration management is a critical aspect of API deployment. **decK**, the Kong automation tool, is utilized to generate and manage Kong declarative configuration files. Starting from the OAS file, decK generates a `kong.yaml` file containing the definitions of Kong services, routes, plugins, and other relevant entities. Validation is performed to ensure that the Kong configuration file is error-free and aligns with the desired setup. Any configuration errors are identified and addressed before deployment.

With the Kong configuration in place, the application is deployed to a Kubernetes cluster using the `kubectl` command. Kubernetes ensures the orchestration and management of the microservices, microgateways, and related components.

6.2.2 Functional Tests

Functional testing is a critical aspect of validating the proposed solutions, encompassing a wide range of authentication and authorization mechanisms, plugins, and API flows. The goal is to ensure that the developed APIs, microservices, and Kong microgateway instances function correctly and securely. Functional tests cover various API flows and scenarios, including:

- **Authentication Mechanisms:** testing different authentication methods such as OAuth 2.0 Authorization Code Flow, OAuth 2.0 Resource Owner Password Credentials Flow, OAuth 2.0 Client Credentials Flow, and SAML-based authentication. These tests validate the ability of consumers to securely authenticate themselves.
- **Authorization:** evaluating authorization mechanisms, including role-based access control (RBAC) and OAuth 2.0 scopes. Tests verify that only authorized users or applications can access specific API endpoints and perform actions.

Moreover, the following aspects are tested and validated:

- **Plugin Testing.** The validation process includes testing the functionality and compatibility of Kong plugins, such as **Rate-Limiting Plugins**, ensuring that rate limiting works as expected to prevent abuse and protect API resources, **Vault Plugins**, validating the integration of Vault plugins for secure credential management and secret retrieval, and more.
- **Error Handling.** Functional tests assess how the system handles errors and edge cases. This includes testing scenarios where users or applications provide incorrect credentials, attempt unauthorized actions, or encounter unexpected issues. Proper error messages and status codes are verified to enhance user experience and security.
- **Data Validation.** Data validation tests are conducted to confirm that data input and output from the APIs adhere to defined specifications and constraints. This helps prevent data corruption, injection attacks, and unexpected behavior.
- **Usability and Integration.** The usability of the APIs and their integration within the larger system are evaluated through functional tests. This involves testing scenarios where APIs are consumed by client applications and ensuring that the APIs interact seamlessly with other system components.

Functional test results are meticulously recorded and reported. This includes capturing details of test scenarios, pass/fail outcomes, error messages, and performance metrics. Reports are made available to developers and stakeholders for review and analysis. The comprehensive functional testing methodologies ensure that the proposed solutions not only work as intended but also adhere to security best practices, industry standards, and user expectations. These tests provide confidence in the robustness and reliability of the developed APIs and microservices.

6.2.3 Performance Tests

Performance testing is essential to determine how well the system handles various levels of load and traffic. While the presented performance tests are simplified for demonstration purposes, they serve as a proof of concept and can be extended for more comprehensive load testing in real-world scenarios.

Performance tests are implemented as **Gatling simulations**. These simulations simulate user interactions with the system, such as making HTTP requests to retrieve *MoviesCatalog* data. The simulations are designed to measure the responsiveness and scalability of the APIs and Kong microgateway under different conditions. For example, simulations may involve a specified number of users concurrently interacting with the system over a defined period. This helps evaluate how the system performs under load and whether it can handle the expected traffic.

Performance test results, including response times and error rates, are captured and published for analysis. While the provided tests are simplified, they lay the foundation for more extensive load and stress testing in real-world scenarios.

6.2.4 Real-Time Monitoring

Observability and monitoring are crucial aspects of system validation. **Prometheus** is used to scrape metrics from Kong microgateway instances, while **Grafana** serves as a dashboard for visualizing these metrics. Developers can monitor the status of the Kubernetes cluster and evaluate the performance of the deployed APIs and microservices in real-time. This allows for proactive detection of issues and ensures that the system operates efficiently and reliably.

6.3 Presentation of the results

This last section collects and shows the results obtained throughout the whole process of developing, deploying and managing APIs, using APIOps best-practices in **GitHub Actions CI/CD workflow for an OAuth 2.0 Code Credentials Flow architecture adopting the API Microgateway Paradigm**. As seen in the previous chapters, a multitude of different scenarios have been presented: some of them uses Jenkins as automation server for CI/CD, others instead, represent different authentication and authorization scenarios, some more are focused on different capabilities, like rate-limiting and more. In pursuit of a comprehensive analysis and validation of the completed work, a specific and thorough use case has been selected. However, as demonstrated in the previous section, the same methodology can be applied to other scenarios.

The considered scenario, however, represents a complete possible workflow that developers may adopt. The scenario has been tested and the architecture has been validated. Starting from this evaluation, it will be possible to easily adapt this scenario to different company's needs.

6.3.1 Description of the scenario

This scenario considered as the example for the validation, shown in figure 6.1 represents a complete Continuous Integration and Continuous Delivery (CI/CD) workflow that developers trigger when new APIs need to be tested, adopting Kong microgateway infrastructural pattern in a Kubernetes cluster. Specifically, an **Azure Kubernetes Service (AKS)** has been installed with an Azure subscription on a Resource Group dedicated to the tests. Kubernetes is used to managed the roll-out of the new deployment and the communication among the pods. In particular, the cluster is composed of the following components:

- An instance of the **APIs microservice** (the *MoviesCatalog* application) along with the **Kong microgateway**. Note that, in a real scenario, multiple microservices with

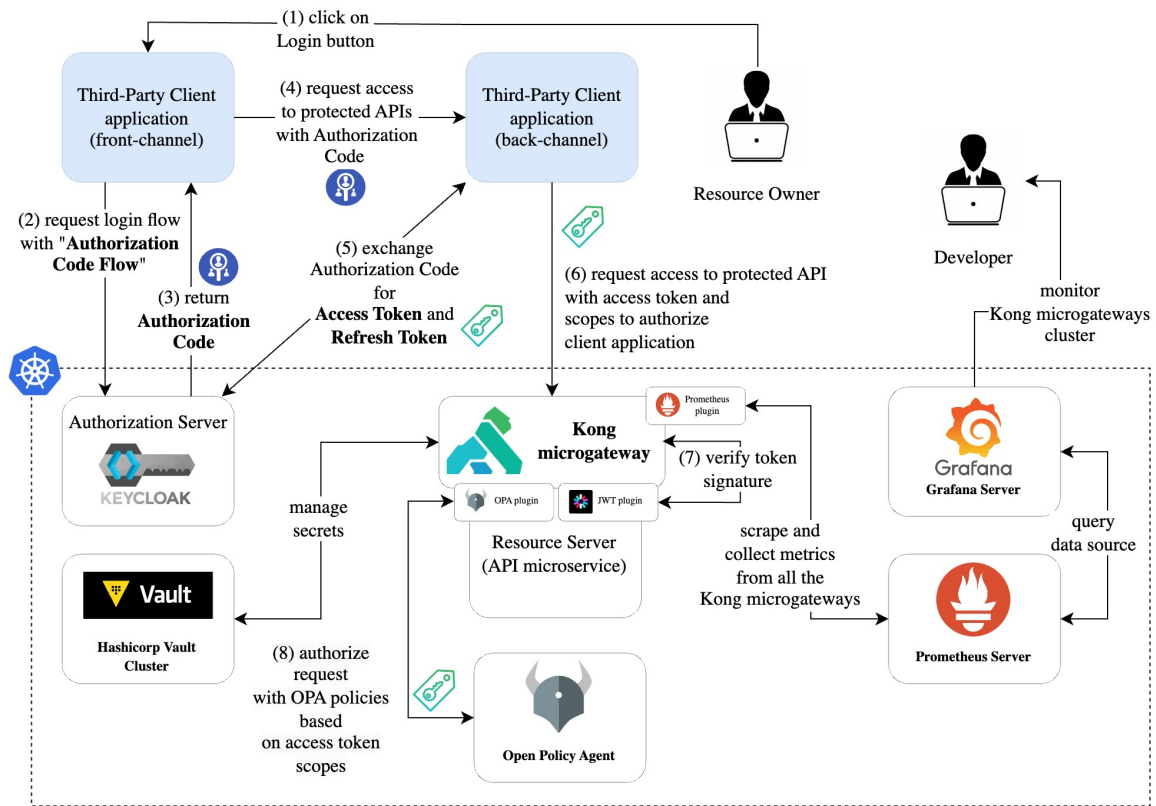


Figure 6.1: Considered scenario in the validation process.

multiple microgateways should be deployed. However, to better show the process and highlight the technologies and techniques used, things are simplified a bit.

- An instance of **Keycloak**, used as Authorization Server in the OAuth 2.0 flow.
- An instance of **Open Policy Agent (OPA) server**, used to authorize requests in the OAuth 2.0 flow.
- An instance of **Prometheus server**, used to scrape metrics from Kong.
- An instance of **Grafana server**, used to display traffic and collected metrics to developers.

Having presented the status of the Kubernetes cluster, let's now suppose that the *MoviesCatalog* application lacks the capability to add a new movie to the catalog. For instance, a new API, performing such operation, should be added to the microservice application:

```

1 POST /movies HTTP/1.1
2 Host: <host>
3 Content-Type: application/json
4 Authorization: Bearer <signed JWT>
5
6 {
7   "name": "The Godfather: Part II",
8   "director": "Francis Ford Coppola",

```



```
9     "genre": "Drama",
10    "releaseyear": 1974,
11    "releasedate": "1974-12-20",
12    "plotsummary": "The continuing saga of the Corleone crime family [...]"
13 }
```

At this point, developers are required to implement such functionality and deploy the new instance of the *MoviesCatalog* application to the test environment, in order to assert the capabilities of the APIs. In particular, the following requirements should be satisfied.

- The **API should be functional**, i.e. it should provides the capabilities required to add a new movie to the catalog (e.g. only validated input should reach the services' logic and be added to the DB).
- The **API should be secure**, i.e. only authenticated consumers should be able to add a new movie to the catalog and no logical issues should be presented once integrated into the system. In this considered scenario, an OAuth 2.0 Code Credentials Flow is set up in order to authenticate and authorize consumers of the API.
- The **API should be efficient**, i.e. once integrated, the system should be able to undergo traffic loads and, eventually, being scaled up to keep serving the API.

As it is presented in this section, all the requirements are satisfied through meticulous attentions and monitoring from the early stage of the development to the deployment into the test environment. Although it is not presented here (as it is not part of this thesis), if the deployment on the test environment satisfies the requirements, it can be considered to be moved to stage or production environments.

A typical workflow (although not the only one, as it depends on the company's policies) that developers are entitled with is the following:

1. Developers opens a **Pull Request** (PR) to integrate the new API's source code on the remote VCS (in this case, github.com). Note that this scenario showcases the usage of the cloud-based GitHub servers managed by the *GitHub, Inc.* organization. Typically, self-managed instances of VCS (e.g. BitBucket, GitLab, GitHub, etc...) are deployed on-premise or on-cloud inside the organization, to have a more fine-grained control over the data.
2. Upon acceptance of the PR, a **Continuous Integration (CI) workflow** is triggered, performing the following operations:
 - (a) **JUnit 5 unit and integration tests** are performed to assert that the implemented code works as expected. **Results are published** to GitHub.
 - (b) The Docker image of the new application is pushed to the remote Docker registry with **Google Jib**.
3. Once the image of the application has been tested and pushed to the registry, a **Continuous Delivery (CD) workflow** is triggered, performing the following operations:
 - (a) The **OpenAPI Specification** file is generated from the source code and audited with **42Crunch Security Assessment scans**. This allows to detect flaws in

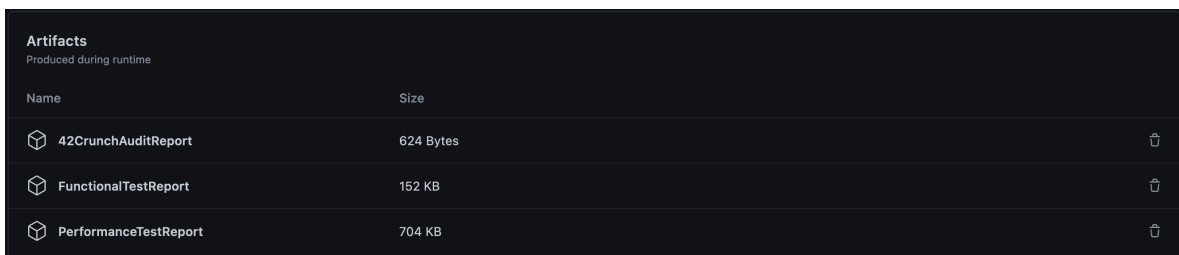
the definition of the API contract. For instance, if the body of the request is not restricted to the specified parameters, 42Crunch returns a failed Security Audits and the pipeline stops.

- (b) The **Kong configuration** file is generated using **deck automation tool**, starting from the OpenAPI Specification file. The file is validated using **deckK** as well, to assert that no errors were made in the integration with Kong plugins, or the definition of Kong consumers, or similar.
 - (c) Customized plugins and additional Kong configurations are rolled out to the **ConfigMap** holding the current configuration in the cluster and the **Kubernetes deployment** of the microservices and microgateways is rolled out too, to apply the new changes.
4. At this point, the new API has been integrated within the cluster, along with the Kong microgateway instance. A series of **Functional Tests** and **Performance Tests** is triggered, evaluating and validating the performance of the new application. This allows developers to assert that the API is secure, following the OAuth 2.0 Code Credentials Flow stated in the requirements, and that the cluster configuration is able to undergo the traffic load.
 5. Developers can **validate the PR** by analyzing the published reports from **Postman**, **Insomnia** and **Gatling**. Moreover, developers can monitor the traffic on the test-environment by analyzing the **Grafana dashboard**.

6.3.2 Results and considerations

At first, let's evaluate the results obtained from the Continuous Integration and Continuous Delivery of the new API. Subsequently, the validation of the architecture is presented, along with the results of the Functional and Performance tests.

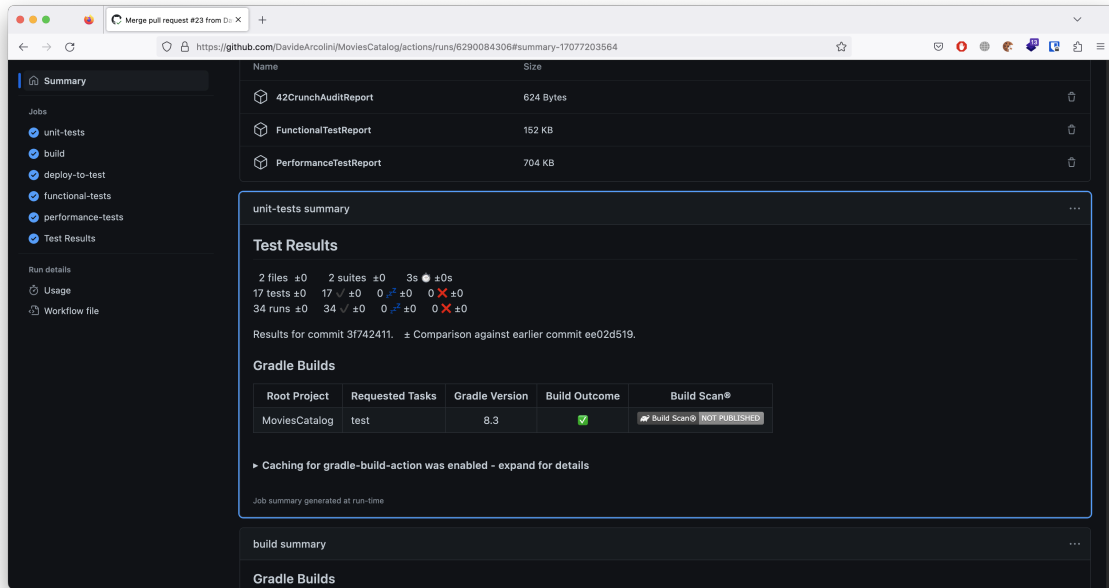
The following is the list of **Artifacts**, that will be later analyzed, generated by this specific workflow:



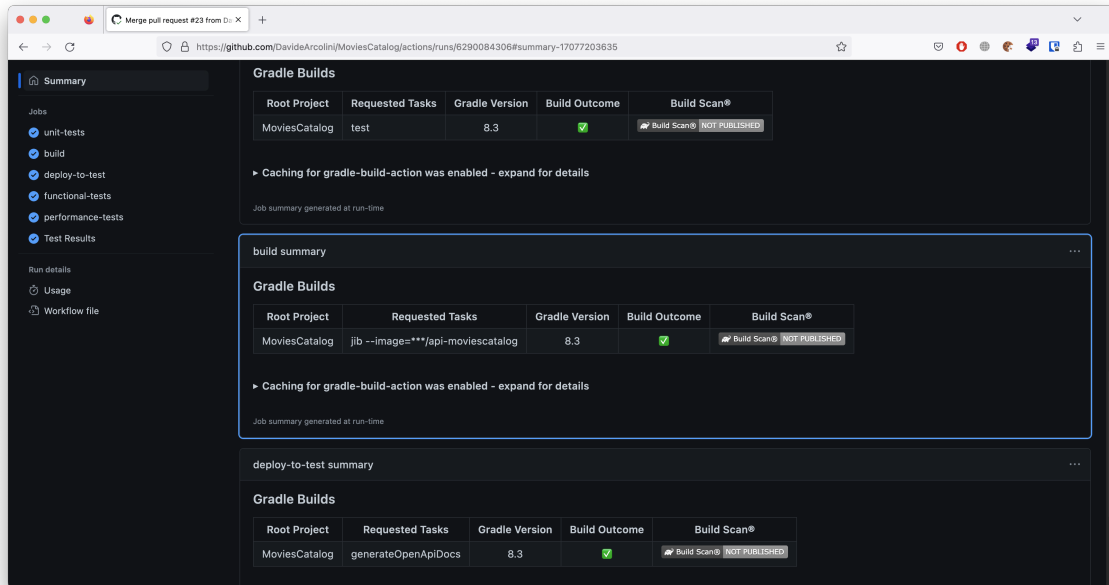
Artifacts	
Produced during runtime	
Name	Size
42CrunchAuditReport	624 Bytes
FunctionalTestReport	152 KB
PerformanceTestReport	704 KB

6.3.2.1 Continuous Integration and Continuous Delivery (CI/CD)

The first step of the workflow, once the PR has been merged into the target branch, is to run a **JUnit 5 suite** of unit tests to assert that the logic of the source code accomplish the results expected and that the application interacts correctly with the persistence entities defined. Unit tests are run with Gradle and results are published as Artifact on GitHub:



At this point, Google Jib builds, tags and pushes the Docker image to the specified Docker registry, ready to be deployed on the K8s cluster. Again, results of the build are published on GitHub:

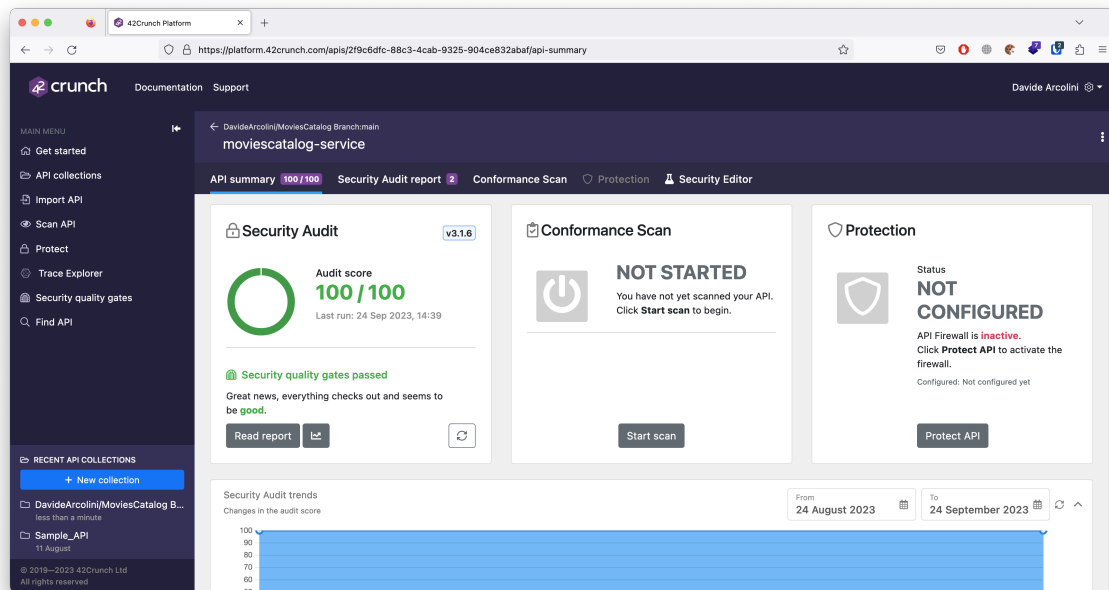


At this point, developers know that the published application image, at least on its own, works as expected. It is time to integrate it within the existing system and validate the architecture. Moving to the Continuous Delivery (CD) steps of the workflow, the first actor encountered is Gradle again. It starts by **generating an OpenAPI Specification** file for

the new application: this is an important piece of information because it will be used by decK to generate the configuration of the API microgateway. For this reason, the OpenAPI Specifications file is sent to the 42Crunch servers, undergoing a Security Assessment scan. An example of log result obtained once this action is completed is the following:

```
1 Audited build/openapi.yaml, the API score is 100
2 No blocking issues found.
3 Details: <url>
4 JSON report was written to: "/github/workspace/42c-report.json"
```

As indicated, once the Security Audit is performed, the result of the assessment is available in the pipeline for the build to proceed or terminate. However, developers need to access the reports of the scan in order to understand what went, eventually, wrong and how to fix it. The result of the assessment can be found on the 42Crunch Platform, under the specific API Collections tab:



The Security Audit report provides insights on:

- The **Security Audit score**, along with the severity of all the issues encountered.
- The **Security Quality Gates** passed, along with all the criteria that have been defined.
- The **Priority Issues** with the suggestions on how to resolve critical problems.

Moreover, the plugins export the `42c-report.json` file that has been declared in pipeline as:

```
1 {
2   "audit": {
3     "report": {
4       "build/openapi.yaml": {
```

```

5         "audited": true,
6         "apiId": "<api_id>",
7         "mode": "discovery",
8         "score": 100,
9         "failures": [],
10        "lifecycle": "updated"
11    }
12  },
13  "deleted": {},
14  "discoveryCollection": {
15    "collectionId": "<collection_id>",
16    "technicalName":
17      ↪ "https://github.com/DavideArcolini/moviescatalog@@main",
18    "name": "DavideArcolini/MoviesCatalog Branch:main"
19  }
20 }

```

Starting from the OpenAPI Specification file, **deck automation tool** is used to generate a `kong.yaml` file containing the definition of the Kong Service entities, Kong Route entities, Kong Plugin entities and all the other entities required. Once the final Kong configuration file is ready, deckK is used to validate it as well:

```
1 deck validate --state kong.yaml
```

The validation **stops the workflow if the file contains any error**. At this point, the deployment to the test environment is performed with `kubectl`, and the new API is integrated in the system, ready to be tested with Functional and Performance assessments.

6.3.2.2 Functional Tests

Functional tests are written to assert that the deployed instance of the service works as expected. Specifically to this scenario, an **OAuth 2.0 Code Credentials Flow** is set up to authenticate and authorize consumers of the new API. The first step to assure that the new API integrates well within the architecture is to **validate it** with functional tests.

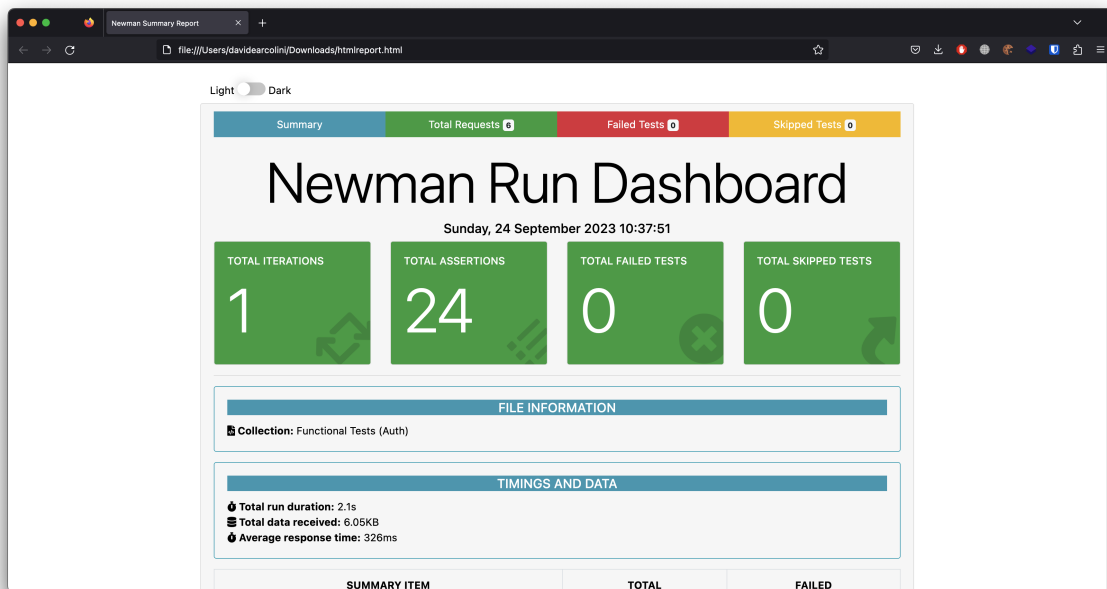
As detailed in the previous chapters, functional tests are written as API collections for Postman or Insomnia Desktop applications and then exported to be run from the pipeline using the corresponding CLI tools: `newman` and `inso`. The tests report is published on GitHub, so that developers can have an idea of what went wrong and why.

In the presented scenario, the following assertions are evaluated:

- Consumer can still interact with the old APIs (e.g., it can retrieve the list of movies in the catalog).
- Consumer fails to add a new movie, if it does not undergo the OAuth 2.0 flow to authenticate itself.
- Consumer fails to add a new movie, if it does not have the required scopes (i.e. `create-movie`) in the Access Token generated with the OAuth 2.0 flow.

- Consumer is able to add a new movie, if it provides a valid Access Token.

Tests are run from the GitHub Action workflow and the report is published as an artifact upon build completion.



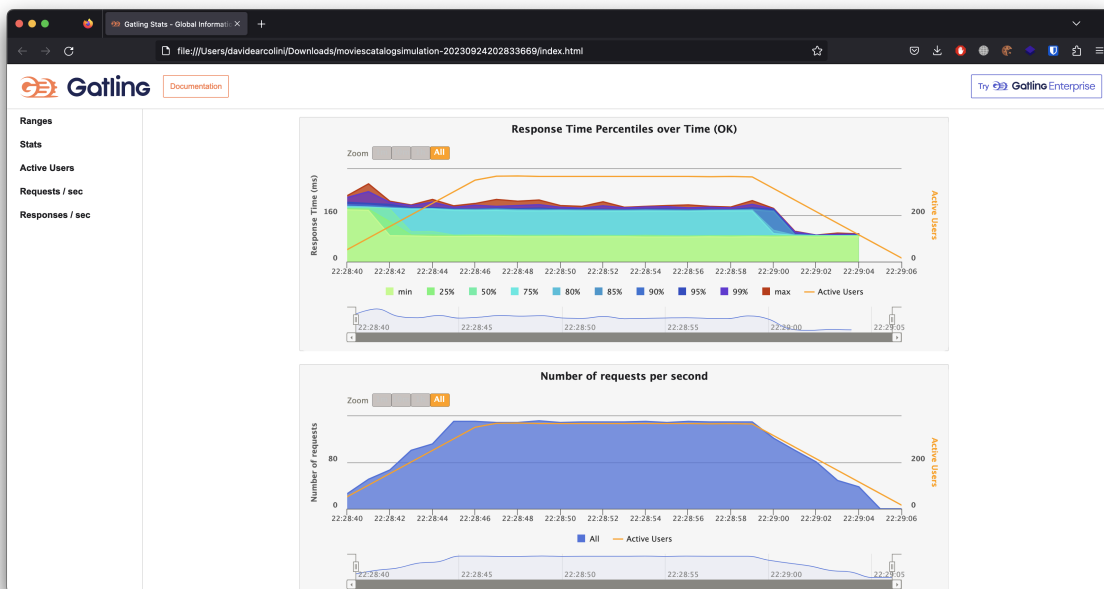
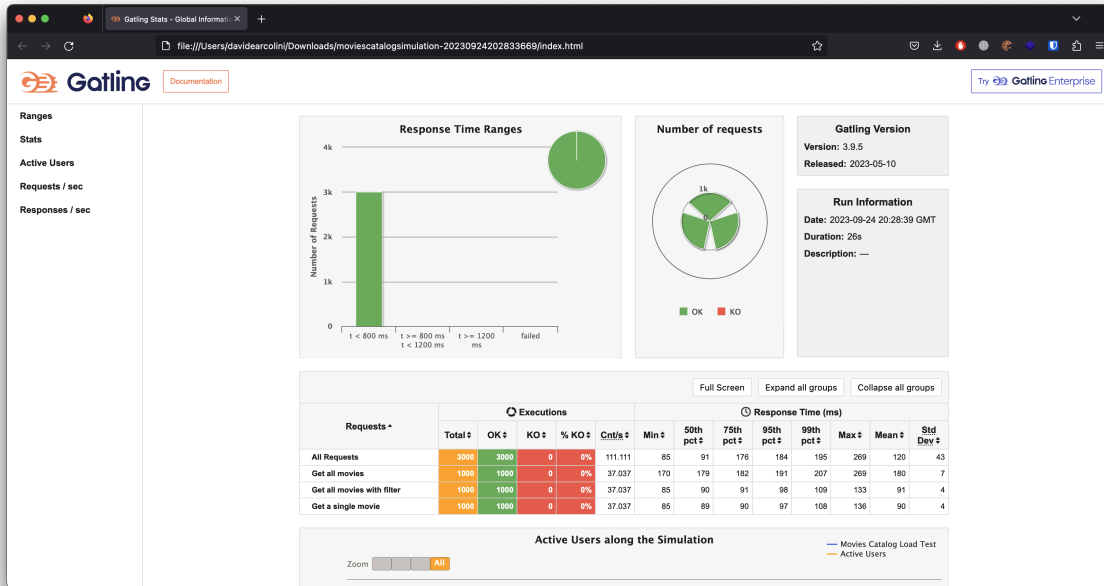
6.3.2.3 Performance Tests

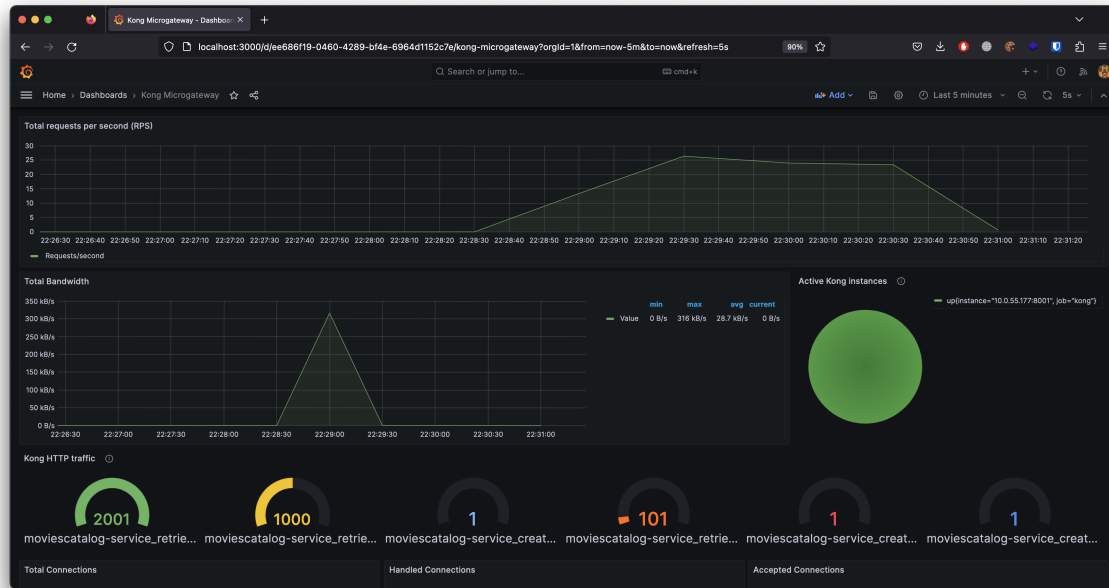
Performance tests are evaluations that assess the responsiveness, reliability, and scalability of an API under various conditions, ensuring it meets **performance expectations and can handle user demands effectively**. As these tests results heavily depend on a variety of factors (e.g. the number of replicas deployed on the cluster, the latency between the automation servers and the exposed APIs, the hardware capabilities of the machine(s) performing the tests, etc...) the presented results are intended to be only a proof of concept. In a real-case scenario, where developers have access to more resources and the traffic is a real concern to the APIs, more rigorous load and stress tests should be written.

As detailed in the previous chapters, performance tests are written as Gatling simulation scenarios which are run with Gradle from the pipeline. The developed scenario per se is pretty straightforward: users make HTTP requests to obtain movies from the catalog. No authentication or authorization takes place, to simplify and make the tests faster. However, the number of users is scaled up arbitrarily to a thousand entities, in order to simulate an example of load traffic for 20 seconds straight (still, 1000 users interacting with the system is not intended to be a production-ready load test. Again, the purpose of this is to create a PoC from which real test-cases can be extracted). The results are published as an artifact upon build completion.

6.3.2.4 Real-Time Monitoring

Developers can refer to the Grafana dashboard to evaluate the performances of the Kong microgateway instances installed and have an idea of the traffic flow in the cluster. The





following images shows the interaction results of the Gatling performance tests, with the 1000 users interacting with the Kong microgateway service.

6.3.3 Analysis of the results

It has been presented a scenario of full-lifecycle API management adopting a Microgateway Infrastructural Pattern with Kong Gateway. Serving web APIs to end-users requires developers and operations teams to test and validate each process that leads to the final product. In the presented scenario, a GitHub Action workflow tests, builds and deploys instances of APIs microservices and Kong microgateways in a test-environment Azure Kubernetes Service (AKS). On the K8s cluster: authentication, authorization and monitoring services are deployed as well. An OAuth 2.0 Code Credentials Flow is tested with Functional tests suites and the capabilities of the architecture is evaluated with Gatling.

As presented in the introductory section of this chapter, the following objectives have been accomplished:

1. Integrate the scenario with **automation tools** to manage the generation of OpenAPI Specifications and the conversion to Kong declarative configuration files. The scenario presented has been integrated in a full-lifecycle API workflow, where Gradle and decK manage the generation, configuration and validation of the OpenAPI Specifications and Kong configurations files.
2. Integrate the scenario with **security assessment tools** for securely developing APIs. The scenario presented has been integrated with security assessment tools, such as 42Crunch, used to test and validate the developed APIs. This and the previous objective serve as a complete pre-deployment configuration phase of the required entities: Kong is automatically and securely configured, allowing developers to have automated results on the API Security Audits and validation of the Kong microgateway instances.
3. Develop production-ready use cases, based on the **authentication and authorization**

industry standards, such as OAuth 2.0 and OpenID Connect. The scenario presented has been integrated with an OAuth 2.0 authentication and authorization flow. Specifically, the workflow proposed ends up building new microservices that are deployed on the K8s cluster along with Keycloak and Open Policy Agent (OPA) servers. This creates the capabilities to test and validate an OAuth 2.0 Code Credentials Flow.

4. Integration with **observability and monitoring tools** to evaluate the performances of the presented scenario. The scenario proposed has been integrated with observability and monitoring tools such as Prometheus and Grafana. These tools allow developers to monitor the cluster status in real-time and evaluate the performances of the deployed APIs.
5. Integration with tools adopting **Infrastructure as Code** (IaC) practices to manage the Kubernetes cluster. The scenario proposed has been integrated with Kubernetes in order to manage the containerization of the microservices. Infrastructure as Code allows developer to define the state of the APIs microservice and Kong microgateway in a `kubernetes-deployment.yaml` file, which is validated and applied in the pipeline to make the new changes in the cluster.

7. Conclusions and Future Work

As Web APIs become more and more important in today’s digital landscapes, this research aims to be an helpful starting point to explore, analyze and further develop the newly defined **API Microgateway Paradigm**. As presented in this study through realistic infrastructural scenarios, the process of exposing APIs to the internet demands substantial effort from organizations delivering services. While security remains a paramount concern, especially when handling end-user information, it represents just one facet propelling the need for this thesis. With the prevailing shift toward microservices architecture, companies seek a means to decentralize the entities governing such infrastructure.

Throughout this thesis, we have explored use cases involving a comprehensive API management infrastructure, integrating widely adopted and cutting-edge technologies like Kubernetes, Keycloak, Open Policy Agent (OPA), Grafana, GitHub Actions, Jenkins, and more. Undoubtedly, the central figure in this research is **Kong Gateway**. As a leader in the market for full-lifecycle API management, the adoption of this tool has naturally followed the course of events. This thesis not only provides a comprehensive overview of the functionalities and capabilities of Kong Gateway but also empowers developers and operations teams to glean valuable insights into the possibilities that arise from adopting Kong Gateway as a microgateway technology.

The objectives set forth in this research have been successfully achieved. The first objective involved defining the API Microgateway Pattern and understanding its integration with Kong Gateway, shedding light on the intricacies of managing Kong Gateway as a microgateway technology. Realistic Proof of Concept scenarios, constituting the second objective, demonstrated the practical implementation of Kong microgateways in enterprise-ready settings, emphasizing the significance of authentication, authorization, and monitoring within Kubernetes environments. The third objective, focused on DevSecOps integration with Kong microgateways, showcased a holistic full-lifecycle API management approach, incorporating automation tools like Jenkins and GitHub Actions to establish secure CI/CD pipelines. The comprehensive testing of the developed PoC scenarios in both on-premise and cloud-based Kubernetes clusters has validated the effectiveness and reliability of Kong microgateways.

However, it is worth noting that there are areas that may benefit from in-depth further considerations. Initially, it is imperative to recognize the multitude of scenarios that warrant consideration. While Keycloak stands out as an exceptionally versatile and widely adopted tool for Identity and Access Management, numerous companies may have embraced, or plan to deploy, alternative technologies (examples include Okta Workforce Identity, Microsoft Entra ID, Auth0, and Ping Identity). The integration of Kong microgateway with these diverse tools necessitates careful consideration by organizations contemplating adoption. Similarly, the integration with Open Policy Agent (OPA) demands heightened attention. OPA’s increasing adoption, particularly due to its alignment with microservices architecture specifications, is

noteworthy. However, the integration showcased in this thesis serves as a basic Proof of Concept, lacking comprehensive insights into High Availability and policy management considerations.

Secondly, Kong Gateway boasts a myriad of plugins that remained unexplored in this research. Each plugin, tailored for specific use cases, requires thorough analysis and testing before seamless integration with Kong microgateway can be ensured. Examples of such plugins encompass IP restrictions, advanced rate-limiting, bot detection, as well as diverse monitoring tools like OpenTelemetry, and logging capabilities such as StatsD or Kafka. These specific use cases present opportunities for further exploration. While the Kong Plugin Hub offers a vast array of capabilities, the consideration of custom plugins, accurately developed for individual use cases, emerges as another possibility deserving dedicated attention.

Finally, as a recap, the following areas present themselves as promising avenues for future research and development:

- Integration of the scenarios presented with different Identity and Access Management (IAM) tools and comparison among them.
- Comprehensive exploration of customized Kong Gateway plugins, in addition to the plugins that have not been considered for this research.
- Scalability and High Availability (HA) considerations for the scenarios.

List of Figures

2.1	Representation of the Full-Lifecycle of the APIs both from the producer and the consumer point of view. [Source: Postman]	4
2.2	Representation of the interaction between a Web Browser and a Web API. [Source: Analytics Vidhya]	6
2.3	Growth percentages of the API traffic per business in 2021. [Source Cloudflare]	6
2.4	Example of Web API request and response.	8
2.5	Schema representing the different types of Web APIs.	9
2.6	A simple CRUD microservice exposing APIs, developed for the purposes of this thesis.	14
2.7	DevSecOps cycle. [Source: Dynatrace]	15
2.8	Example of APIOps approach using GitLab as VCS and Jenkins as Automation Server.	16
2.9	CI/CD most common steps to integrate, test and deploy applications.	17
2.10	42Crunch interface. [Source: 42Crunch]	19
3.1	Abstraction overview of a general architecture developed adopting a microgateway infrastructural model with Kong Gateway.	20
3.2	General architecture of a retail application adopting a microservices architecture model.	21
3.3	Microservices architecture forecast market growth by Market Research at 2023. [Source: Market Research]	23
3.4	General architecture of a monolithic application.	24
3.5	Strangler Pattern allows monolithic applications to shrink down to microservices in a controlled environment.	25
3.6	Hexagonal software architecture adapted to a microservice implementation.	26
3.7	Kubernetes architecture [Source: Cloud Native Computing Foundation].	28
3.8	Example of Kubernetes cluster used to manage the deployments of two microservices, each of which has 3 replicas deployed as pod. Microgateways are deployed close to the microservices, inside the same pod. The picture does not comprehend all the other entities involved in the development of this thesis (such as: K8s services, Databases, Vaults, IdP, etc...).	29
3.9	Traditional and monolithic API gateway proxying all the requests to the correspondent microservice.	30
3.10	Simplified example of a banking system implementing a Facade Pattern to manage the banking subsystems. The similarities between a facade and an API gateway can be easily seen in the diagram.	32
3.11	Simplified comparison between traditional and monolithic API gateways versus modern and innovative microgateways developed for this thesis.	33

3.12	Example of configuration of entities in API Microgateway Pattern. The microgateway is lightweight, as it stores the configuration in memory (no DB required). It is rapid in being configured and mounted next to the microservice. And it is flexible as it externalize the supporting businesses to other entities. . .	34
3.13	Table summarizing the differences between the traditional API gateway and the presented API microgateways.	35
3.14	Overview of Kong Gateway as traditional API Gateway. Requests originating from an API client enter the Gateway, undergo modification and management by the proxy according to the Gateway's configuration, and then get forwarded to upstream services. [Source: Kong]	36
3.15	Gartner Magic Quadrant(s) for Full-Lifecycle API Management	37
3.16	Overview of Kong microgateway terminology. Clients, which may be associated to a consumer, make requests based on the configured routes. The request is mapped to the specific service and proxied to the correspondent upstream. . . .	39
3.17	Three different example of Kong microgateway adapted to three different microservices. Kong Gateway instance remains the same, but the declarative configuration defined in CI/CD loads different plugins for each use case.	42
3.18	Adapting decK to the API Microgateway Pattern . Developers can now keep microgateway configurations under Version Control in the microservice-dedicated Git repository. Each configuration is tailored to the microservice and decK manages the integration and validation of the Kong declarative Config. . .	45
4.1	Example of a complete enterprise-ready scenario. Kong microgateway is integrated within an OAuth 2.0 Authorization Code Flow adopting Keycloak as the Authorization Server and OPA as the policy enforcement framework. HashiCorp Vault integration is adopted to manage confidential secrets, while Prometheus and Grafana monitors traffic flow in the cluster.	49
4.2	OpenAPI Specification overview of the <i>MoviesCatalog</i> application.	52
4.3	Scenario implementing the integration within Kong microgateway and HashiCorp Vault.	59
4.4	HashiCorp Vault User Interface (UI) to add a new secret.	60
4.5	Scenario implementing the integration within Kong microgateway, Prometheus and Grafana.	63
4.6	Prometheus architecture [Source Prometheus].	64
4.7	Kong (Official) Grafana Dashboard [Source Grafana].	66
4.8	An Identity and Access Management (IAM) infrastructure has been developed in these PoC to test Kong microgateway capabilities to implement authentication and authorization flows.	68
4.9	Scenario implementing the integration within Kong microgateway, Keycloak and Open Policy Agent.	69
4.10	Keycloak Single Sign-On (SSO) interface for granting access to Client applications.	71
4.11	Enforcement of policies with OPA.	72
4.12	Scenario implementing the OAuth 2.0 Authorization Code Flow in Kubernetes with Kong microgateway.	81
4.13	Insomnia desktop client simulating the exchange of the Authorization Code for the Access Token.	82
4.14	Scenario implementing the OAuth 2.0 Client Credentials Flow in Kubernetes with Kong microgateway.	83
4.15	Insomnia desktop client simulating the access to the API with the Access Token.	84

4.16	Scenario implementing the OAuth 2.0 Resource Owner Password Flow in Kubernetes with Kong microgateway.	85
4.17	Scenario implementing the OpenID Connect Authorization Code Flow in Kubernetes with Kong microgateway.	86
4.18	Sequence Diagram showing the interaction between the end-user, Kong microgateway and the IdP. OPA integration is missing. [Source: Kong OpenID Connect]	87
5.1	The three stages considered in the CI/CD approach developed for this thesis.	91
5.2	API Operations infrastructure and processes in CI/CD pipelines.	93
5.3	Unit and Integration Phase for the CI steps of the pipelines.	96
5.4	APIs and Kong Microgateway Management Phases for the CI steps of the pipelines.	96
5.5	Deployment and Test Phase for the CD steps of the pipelines.	103
5.6	Test suite containing functional tests designed in Insomnia.	106
6.1	Considered scenario in the validation process.	115

Bibliography

- [GN47] Herman H. Goldstine and John von Neumann. “Planning and coding of problems for an Electronic Computing Instrument”. In: 1947. URL: <https://api.semanticscholar.org/CorpusID:60716769>.
- [Con68] Melvin E. Conway. “How Do Committees Invent?” In: *Datamation* (Apr. 1968). URL: <http://www.melconway.com/research/committees.html>.
- [CG68] Ira W. Cotton and Frank S. Grestorex. “Data Structures and Techniques for Remote Computer Graphics”. In: *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*. AFIPS ’68 (Fall, part I). San Francisco, California: Association for Computing Machinery, 1968, pp. 533–544. ISBN: 9781450378994. DOI: [10.1145/1476589.1476661](https://doi.org/10.1145/1476589.1476661). URL: <https://doi.org/10.1145/1476589.1476661>.
- [Mal90] C. Malamud. *Analyzing Novell Networks*. VNR computer library. Van Nostrand Reinhold, 1990. ISBN: 9780442003647. URL: https://books.google.it/books?id=_vFSAAAAMAAJ.
- [Fie00] Roy Fielding. “Architectural Styles and the Design of Network-based Software Architectures”. PhD thesis. 2000. URL: <https://ics.uci.edu/~fielding/pubs/dissertation/top.htm>.
- [Rod05] Peter Rodgers. “Service-Oriented Development on NetKernel- Patterns, Processes & Products to Reduce System Complexity”. In: *Web Services Edge 2005 East: CS-3*. CloudComputingExpo. 2005. URL: <https://web.archive.org/web/20180520000000/http://www.cloudcomputingexpo.com:80/general/sessions/cs3.htm> (visited on 07/03/2017).
- [Har12] Dick Hardt. *The OAuth 2.0 Authorization Framework*. RFC 6749. Oct. 2012. DOI: [10.17487/RFC6749](https://www.rfc-editor.org/info/rfc6749). URL: <https://www.rfc-editor.org/info/rfc6749>.
- [Ste14] Jan Stenberg. “Exploring the Hexagonal Architecture”. In: *InfoQ* (Oct. 31, 2014). URL: <https://www.infoq.com/news/2014/10/exploring-hexagonal-architecture/> (visited on 08/12/2019).
- [CMJ15] Brian Campbell, Chuck Mortimore, and Michael B. Jones. *Security Assertion Markup Language (SAML) 2.0 Profile for OAuth 2.0 Client Authentication and Authorization Grants*. RFC 7522. May 2015. DOI: [10.17487/RFC7522](https://www.rfc-editor.org/info/rfc7522). URL: <https://www.rfc-editor.org/info/rfc7522>.
- [Nic17] Saverio Giallorenzo et. al. Nicola Dragoni. “Microservices: Yesterday, Today, and Tomorrow”. In: *Present and Ulterior Software Engineering*. 2017, pp. 195–216. ISBN: 978-3-319-67424-7. DOI: [10.1007/978-3-319-67425-4_12](https://doi.org/10.1007/978-3-319-67425-4_12).
- [Ric18] C. Richardson. *Microservices Patterns: With examples in Java*. Manning, 2018. ISBN: 9781617294549. URL: <https://books.google.it/books?id=UeK1swEACAAJ>.
- [Ton18] Moritz Steiner Tony Lauro Martin Flack. “State of the internet / Retail attacks and API traffic”. In: *Akamai* (2018). URL: <https://www.akamai.com/site/it/>

- [documents/state-of-the-internet/state-of-the-internet-security-retail-attacks-and-api-traffic-report-2019.pdf](#).
- [Pao20] Mark O’Neill et. al. Paolo Malinverno Kimihiko Iijima. *Gartner Magic Quadrant for Full Life Cycle API Management*. Tech. rep. Sept. 2020. URL: <https://www.gartner.com/en/documents/3990768>.
- [Res20] Market Research. *Cloud Microservices Market 2020 Trends, Market Share, Industry Size, Opportunities, Analysis and Forecast by 2026 – Instant Tech Market News*. 2020. URL: <https://www.fortunebusinessinsights.com/cloud-microservices-market-107793>.
- [Hec21] Melissa van der Hecht. “What is APIOps”. In: *Kong Inc.* (Feb. 25, 2021). URL: <https://konghq.com/blog/enterprise/what-is-apiops>.
- [Roy21] Winston Royce. “Managing the Development of Large Software Systems (1970)”. In: Feb. 2021, pp. 321–332. ISBN: 9780262363174. DOI: [10.7551/mitpress/12274.003.0035](https://doi.org/10.7551/mitpress/12274.003.0035).
- [Sha21] Mark O’Neill et. al. Shameen Pillai Kimihiko Iijima. *Gartner Magic Quadrant for Full Life Cycle API Management*. Tech. rep. Sept. 2021. URL: <https://www.gartner.com/en/documents/4006268>.
- [FNR22] Roy T. Fielding, Mark Nottingham, and Julian Reschke. *HTTP Semantics*. RFC 9110. June 2022. DOI: [10.17487/RFC9110](https://doi.org/10.17487/RFC9110). URL: <https://www.rfc-editor.org/info/rfc9110>.
- [Mic22] Microsoft. *.NET Microservices Architecture for Containerized .NET Applications*. Sept. 2022. URL: <https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/direct-client-to-microservice-communication-versus-the-api-gateway-pattern>.
- [Sha22] Mark O’Neill et. al. Shameen Pillai Kimihiko Iijima. *Gartner Magic Quadrant for Full Life Cycle API Management*. Tech. rep. Nov. 2022. URL: <https://www.gartner.com/en/documents/4021131>.
- [Dum23] Paul Dumas. *Reference Model for API Management Solutions*. Tech. rep. Mar. 2023. URL: <https://www.gartner.com/en/documents/4208899>.

Sitography

- [Fow04] Martin Fowler. *Strangler Pattern*. 2004. URL: <https://martinfowler.com/bliki/StranglerFigApplication.html>.
- [Age23a] Open Policy Agent. *OPA REST API*. 2023. URL: <https://www.openpolicyagent.org/docs/latest/rest-api/>.
- [Age23b] Open Policy Agent. *Open Policy Agent*. 2023. URL: <https://www.openpolicyagent.org/docs/latest/>.
- [Aut23] Auth0. *Introduction to Identity and Access Management (IAM)*. 2023. URL: <https://auth0.com/docs/get-started/identity-fundamentals/identity-and-access-management#iam-basic-concepts>.
- [23a] Corretto. 2023. URL: <https://aws.amazon.com/corretto/>.
- [Dev23] Facebook Developers. *Facebook Graph API Documentation*. 2023. URL: <https://developers.facebook.com/docs/graph-api>.
- [Ere23] Paulo Silva Erez Yallon Inon Shkedy. *OWASP API Security Top Ten (2023 Edition)*. 2023. URL: <https://owasp.org/API-Security/editions/2023/en/0x00-header/>.
- [23b] Gatling. 2023. URL: <https://gatling.io/docs/gatling/>.
- [23c] Gatling Gradle Plugin. 2023. URL: https://gatling.io/docs/gatling/reference/current/extensions/gradle_plugin/.
- [23d] Jib. 2023. URL: <https://cloud.google.com/java/getting-started/jib>.
- [Ken23] David Saff et. al. Kent Beck Erich Gamma. *JUnit 5*. 2023. URL: <https://junit.org/junit5/>.
- [Key23] Keycloak. *Keycloak Server Administration Guide*. 2023. URL: https://www.keycloak.org/docs/latest/server_admin/index.html.
- [Kon23a] Kong. *Admin APIs*. 2023. URL: <https://docs.konghq.com/gateway/3.4.x/admin-api/>.
- [Kon23b] Kong. *DB-less and Declarative Configuration*. 2023. URL: <https://docs.konghq.com/gateway/3.4.x/production/deployment-topologies/db-less-and-declarative-config/>.
- [Kon23c] Kong. *deck*. 2023. URL: <https://docs.konghq.com/deck/latest/>.
- [Kon23d] Kong. *deck*. 2023. URL: <https://docs.konghq.com/deck/1.27.x/guides/apiops/>.
- [Kon23e] Kong. *deck*. 2023. URL: <https://docs.konghq.com/deck/1.27.x/guides/apiops/#configuration-generation>.
- [Kon23f] Kong. *deck*. 2023. URL: <https://docs.konghq.com/deck/1.27.x/guides/apiops/#configuration-transformations>.
- [Kon23g] Kong. *deck*. 2023. URL: <https://docs.konghq.com/gateway/latest/production/environment-variables/>.
- [Kon23h] Kong. *Kong Gateway*. 2023. URL: <https://docs.konghq.com/gateway/latest/>.

- [Kon23i] Kong. *OpenID Connect (OIDC) plugin*. 2023. URL: <https://docs.konghq.com/hub/kong-inc/openid-connect/>.
- [Kon23j] Kong. *Prometheus*. 2023. URL: <https://docs.konghq.com/hub/kong-inc/prometheus/>.
- [Kon23k] Kong. *Proxy Reference*. 2023. URL: <https://docs.konghq.com/gateway/3.4.x/how-kong-works/routing-traffic/>.
- [Kon23l] Kong. *Secrets Management*. 2023. URL: <https://docs.konghq.com/gateway/latest/kong-enterprise/secrets-management/>.
- [Kon23m] Kong. *Security Assertion Markup Language (SAML) plugin*. 2023. URL: <https://docs.konghq.com/hub/kong-inc/saml/>.
- [23e] *Kubernetes*. 2023. URL: <https://kubernetes.io/>.
- [Mol23] Daniele Molteni. *Landscape of API traffic*. Sept. 2023. URL: <https://blog.cloudflare.com/landscape-of-api-traffic/>.
- [42C] 42Crunch. *42Crunch*. URL: <https://42crunch.com/>.
- [IBM] IBM. *DevSecOps*. URL: <https://www.ibm.com/topics/devsecops>.
- [Lie] Shannon Lietz. *DevSecOps Manifesto*. URL: <https://www.devsecops.org/>.
- [Sma] SmartBear. *OpenAPI Specification*. URL: <https://swagger.io/specification/>.