

POLITECNICO DI TORINO

Master's Degree in Computer Engineering



**Politecnico
di Torino**

Master's Degree Thesis

Zero Trust Architectures

Supervisor

Prof. Riccardo SISTO

Ivan AIMALE, Blue Reply

Candidate

Andrea SCOPPETTA

October 2022

*To everyone that stayed with me along the path
and made it a wonderful adventure.*

Table of Contents

List of Figures	VI
Acronyms	IX
1 Introduction	1
2 Background concepts	3
2.1 Cloud Computing	3
2.2 Advantages and disadvantages of Cloud Computing	5
2.3 Service Mesh	6
2.4 Istio	8
2.5 Zero Trust Model	14
3 Design of the Proof of Concept	16
3.1 Cluster creation	17
3.2 Sample application development	17
3.3 Service Mesh installation and configuration	22
3.4 Application exposition	28
3.5 Authorization server installation and configuration	30
3.6 External services access	32
4 Implementation of the Proof of Concept	36
4.1 EC2 machine creation	36
4.2 Minikube installation	37
4.3 Istio inside Minikube installation	37
4.4 Development and deployment of BookInfo application	38
4.5 Ingress Gateway and end-users policies implementation	43
4.6 Intra-services policies implementation	47
4.7 MetallB configuration	48
4.8 Nginx configuration	48
4.9 Keycloak configuration and deployment	49

4.10 Egress Gateway configuration	51
5 PoC results validation	54
5.1 Zero Trust paradigm correctness	54
5.2 Ingress Gateway correctness	59
5.3 Egress Gateway correctness	60
5.4 Keycloak correctness	62
6 Real case scenario: Upgrading to a Service Mesh	64
6.1 Components description	64
6.2 Architecture description	65
6.3 Azure	66
6.4 IAM on-premise	67
6.5 Architectural flaws	68
6.6 Upgraded architecture	68
7 Real case results validation	70
7.1 Zero Trust paradigm correctness	70
7.2 Ingress Gateway correctness	72
7.3 Egress Gateway correctness	73
8 Conclusions and future works	75
Bibliography	77

List of Figures

2.1	Summary of service models	4
2.2	Service Mesh structure	7
2.3	Proxies communication example	7
2.4	Istio Security Architecture example	9
2.5	Peer Authentication example	11
2.6	Request Authentication example	12
2.7	Virtual Service example	13
2.8	Zero Trust Principles	14
3.1	PoC Layers	20
3.2	BookInfo Application Structure	20
3.3	Get book details - Sequence diagram	21
3.4	Post new book details - Sequence diagram	22
3.5	Post new review - Sequence diagram	22
3.6	Details service Authorization Policy	27
3.7	BookInfo structure using Istio	28
3.8	PoC Intermediate Structure	29
3.9	PoC with Keycloak	32
3.10	Egress Gateway example	33
3.11	PoC Final Structure	35
4.1	Kiali dashboard	37
4.2	Product Page products controller	39
4.3	Product Page Dockerfile	40
4.4	Details Deployment YAML	41
4.5	Details Service YAML	42
4.6	Details Service Account YAML	43
4.7	Ingress Gateway YAML	44
4.8	Ingress Gateway Virtual Service YAML	45
4.9	Ingress Gateway Request Authentication YAML	46
4.10	Authorization Policy for Admins - GET	47

4.11	Peer Authentication - STRICT mode	48
4.12	Reverse Proxy configuration	49
4.13	Keycloak dashboard	49
4.14	Postman REST request to obtain JWT	50
4.15	Keycloak Service YAML	50
4.16	Egress Gateway YAML	51
4.17	Egress Gateway Virtual Service YAML	52
4.18	WorldtimeAPI Service Entry	53
5.1	Encrypted response example	55
5.2	Clear text response example	55
5.3	Service to Service Authentication steps	56
5.4	Service to Service Authorization steps	56
5.5	Least privilege example	57
5.6	Least privilege test	57
5.7	Least privilege test part 2	58
5.8	Prometheus dashboard example	58
5.9	Custom observability pipeline example	59
5.10	Ingress Gateway logs - Authorization policy example	60
5.11	Error 403 test	60
5.12	External service test	61
5.13	Egress Gateway - External service logs	62
5.14	Decoded JWT example	63
6.1	User Account Logic Architecture	66
6.2	User Account Technical Architecture	66
6.3	Logic Architectures using Istio	69
7.1	Clear text response example	71
7.2	Encrypted response example	71
7.3	Least privilege test	72
7.4	Curl success test	72
7.5	Ingress Gateway logs	73
7.6	Error 403 test	73
7.7	Curl on-premise service	73
7.8	Egress Gateway logs	74

Acronyms

API

Application Programming Interface

AWS

Amazon Web Services

CA

Certification Authority

DB

Database

EC2

Elastic Compute Cloud

HTTP

Hypertext Transfer Protocol

IaaS

Infrastructure as a Service

IAM

Identity and Access Management

IT

Information Technology

JWKS

JSON Web Key Set

JWT

JSON Web Token

mTLS

Mutual Transport Layer Security

OIDC

OpenID Connect

PaaS

Platform as a Service

PoC

Proof of Concept

SaaS

Software as a Service

SQL

Structured Query Language

SSO

Single Sign On

TLS

Transport Layer Security

VM

Virtual Machine

VPN

Virtual Private Network

Chapter 1

Introduction

In the last two decades, public cloud has taken hold more and more so that an increasing number of companies has decided to migrate their workflows from their own private datacenters to a public cloud provider. Application development needs to be more adaptable as we transition to cloud solutions, creating a new cloud native approach consisting of microservice instead of monoliths. This led to important security challenges such as **workload authentication**. Following the cloud's growth, new tools and models arose, like the **Service Mesh** and the **Zero Trust paradigm**. The first one is a dedicated infrastructural layer that can be added to applications, allowing to transparently add capabilities like observability, traffic management, and security, without adding them to the application's code. The second one is an IT security approach that assumes that no network perimeter is safe so every communication must be authenticated. The idea behind it is that security must be developed with the strategy *"Never trust, always verify"* in mind because implicit trust is always a risk.

With these concepts in mind, the goal of the thesis is to understand how a *Zero Trust architecture* can be designed, which PROs and CONs it may have and in which cases it can be applied, especially in the enterprise environment.

To dig deeper, a **Proof of Concept** was realized to test whether a Zero Trust architecture can be achieved using a Service Mesh, in particular a product called **Istio**. It consisted in a sample **Spring Boot** application deployed in a Kubernetes cluster created using **Minikube**. On top of the application, Istio was installed and configured to fulfill the Zero Trust model. Furthermore, Istio's additional features such as the **Ingress Gateway** and the **Egress Gateway** were used to obtain the highest security level possible. A simple authentication server was included in the PoC as well, realized using **Keycloak**. After realizing the whole architecture, a series of tests were carried out to verify whether the Zero Trust model and the security requirements were achieved or not. The results showed that a Service Mesh **could be used** to achieve a Zero Trust architecture, so the focus shifted on

a real case scenario: implementing the Service Mesh in an already up and running application, that provides a critical workflow in a big company.

This was possible thanks to the help of Blue Reply, my internship company. The application already implemented the Zero Trust model without the use of the Service Mesh but this implementation had some flaws and disadvantages like: an excess of utility code in the microservices, no observability tools and little maintainability. Its implementation involved the use of a JWT exchange from the original user request throughout the whole transaction. After shutting this feature down, implementing and configuring Istio, a series of tests were carried out to test the improvements compared to the initial structure. The results confirmed the initial intentions so we can affirm that the Service Mesh is a very useful tool to obtain a high level of security and observability even in an already running application.

The following chapters will talk about the two main parts of the thesis: the Proof of Concept and the real case scenario. They will help us understand how a Zero Trust architecture can be achieved thanks to the Service Mesh and which may be the guidelines to implement it.

In the first part I'll go into details of how the Proof of Concept was designed and implemented in all its parts: the cloud environment in which it's deployed, the sample application used to test the Service Mesh, and Istio's installation and configuration, which is the main focus. At the end, various tests are reported to prove the validity of what has been done before. The second part is about the real case scenario. Its architecture and the original way it implemented the Zero Trust paradigm will be described and explored as well as the flaws it originally had. Then, I'll move onto describing what changes have been done and how they affected and improved the original architecture. Finally, like in the case of the PoC, various test will be reported to confirm that implementing the Service Mesh was a good choice and served the purpose of creating a Zero Trust architecture even in an already up and running enterprise application.

Chapter 2

Background concepts

The goal of this chapter is to give a brief explanation of the background concepts of the thesis. It will talk about Cloud Computing and its diffusion now days, focusing also on its security challenges. Then it will introduce the concept of Service Mesh and how it can be exploited to mitigate security risks in a cloud environment.

2.1 Cloud Computing

Delivering hosted services through the internet is referred to as "cloud computing" in general. It can be mainly divided into three categories: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS).

- *IaaS*: Infrastructure as a Service is supplied by cloud providers such as Amazon Web Services (AWS), Google Cloud Platform or Microsoft Azure. They provide virtual machines or other kinds of services, usually accessible by simple browser. The company owns the physical server but the problems about scaling, load balance and firewalls are in charge of the customer.
- *PaaS*: In the Platform as a Service model development tools are hosted on cloud service providers' infrastructures. Using APIs, web portals, or gateway software, users can access these tools online. PaaS is utilized for the creation of all types of software, and numerous PaaS service providers host the finished product. Salesforce's Lightning Platform, AWS Elastic Beanstalk, and Google App Engine are examples of popular PaaS platforms.
- *SaaS*: Software as a Service is a method of software delivery that made available web services, or software applications, across the internet. Users can use a PC or mobile device with internet connectivity to access SaaS applications and services from any location. They can also access to databases and application

software under the SaaS model. The productivity and email capabilities provided by Microsoft 365 are a typical example of a SaaS application.

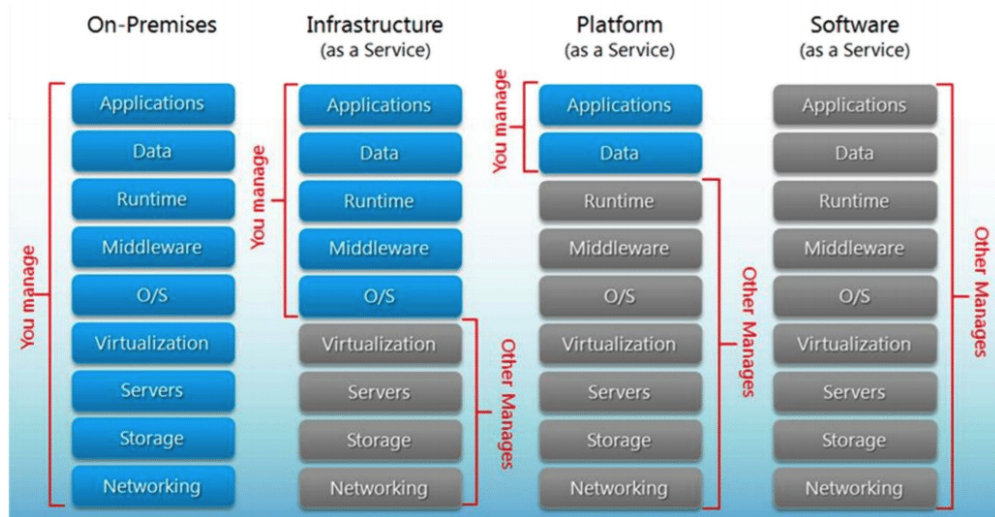


Figure 2.1: Summary of service models

A cloud can be deployed in multiple ways:

- *Public*: Is provided by a so called cloud provider, such as Amazon, Microsoft or IBM, and it's accessible by everyone through a subscription or a pay-to-use format.
- *Private*: Is hosted in a proprietary network or a data center that provides services with specific access and authorization settings to a small group of users.
- *Hybrid*: is made with both public and private resources. In terms of legacy or security, this may be quite useful.

Additionally, enterprises are adopting a multi-cloud approach more frequently, which involves using multiple IaaS providers. This makes it possible for workloads to move between several cloud providers or even run simultaneously across two or more of them.

It heavily relies on virtualization and automation in order to get abstraction and easy provisioning and to obtain the minimum effort from IT staff. These two concepts are tied together by the concept of *orchestration* which provides an high degree of scalability, fault resilience.

Lately, virtualization is made by containers which are essentially lightweight virtual machines, without a full fledged operating system, that are usually used

in a microservice architecture. They allow efficient orchestration since they can be started up quite quickly, they can be easily replaced since their state is usually backed in a persistent external database and they allow horizontal scaling to handle more requests as they grow in number.

2.2 Advantages and disadvantages of Cloud Computing

First of all we'll describe it's characteristics, which will then be analysed to point out the advantages and the disadvantages.

Elasticity is one of the main characteristics, companies can scale up or down the resources based on the needs and this goes along with the *Pay-per-Use* billing mode. In this way companies pay only for the exact amount of resources they use.

Cloud providers ensure *resilience* due to an high number of servers, usually spanning across multiple regions, allowing redundancy and ensuring that workloads are always up and running.

Furthermore, it's fairly easy to move workloads on or off the cloud, or to move from a cloud provider to another, allowing *migration* for cost savings or to use new services as they roll out. In conclusion, it is designed to support *multi-tenancy*, which enables a large number of users to share the same physical infrastructures or applications while maintaining the confidentiality and privacy of their own data.

All of these characteristics, as we can imagine, lead to multiple important advantages for modern businesses:

- *Cost management*: Companies don't have to buy and maintain their own datacenters, cutting the costs of the hardware, the bills, and the whole infrastructure as well as the cost of an IT staff to maintain everything. Additionally, the companies can grow without worrying about enlarging their IT resources, they just have to ask for more to their cloud providers. Cloud computing also cuts costs related to downtime, since it rarely happens thanks to the reasons told before.
- *Data and workload mobility*: Every resource stored in the cloud can easily be accessed through the internet by every device. This enables remote employees to stay up to date with co-workers and customers. It also makes it simple for end users to process, store, retrieve, and restore resources. Additionally, cloud vendors immediately supply all upgrades and updates, saving time and effort.
- *Business continuity and disaster recovery (BCDR)*: Thanks to the intrinsic infrastructure of the cloud, organizations don't need to worry about data loss. Even if the user's device is inoperable all the data is safe in the cloud,

safely stored, redundantly, by the providers that have well structured disaster recovery techniques. This is important to BCDR and ensures that workloads and data are accessible even in the event of damage or disruption to the organization.

Despite all the benefits described, cloud computing creates new challenges:

- *Cloud security*: This is also the main focus of the thesis. Organizations who rely on the cloud run the risk of data breaches, API and interface hacks, compromised passwords, and authentication problems. Security necessitates paying close attention to corporate practices, cloud setups, and policy. In particular, the thesis is more focused on the intra-cluster security.
- *IT governance*: Since there is no control over the provisioning, deprovisioning, and management of infrastructure operations, the cloud computing model's emphasis on do-it-yourself capabilities can make IT governance challenging. It may be difficult to manage risks and security, IT compliance, and data quality effectively as a result.
- *Cloud performance*: The organization purchasing cloud services from a provider has little control over performance factors like latency. If firms do not have backup plans in place, network and provider disruptions can hinder productivity and interfere with business operations.
- *Vendor lock-in*: Changing cloud providers can frequently result in serious problems. Technical incompatibilities, legal and regulatory restrictions, and high expenses associated with big data migrations are some examples of this.

2.3 Service Mesh

Modern applications are built as a collection of microservices deployed altogether on a cluster, this allows to apply the separation of concerns paradigm, to improve modularity and performance optimization.

The drawback of this architecture is that it might be challenging to manage as it increases in size and complexity. Service discovery, load balancing, failure recovery, metrics, and monitoring are just a few examples of its requirements. This is where the *Service Mesh* comes in handy. It is a dedicated infrastructure layer that can be added to the applications, allowing to transparently add capabilities like observability, traffic management, and security, without adding them to its code. It is placed at the level of service to service communications through the use of a proxy.

Basically, a Service Mesh is made by a *Control Plane* and a *Data Plane*.

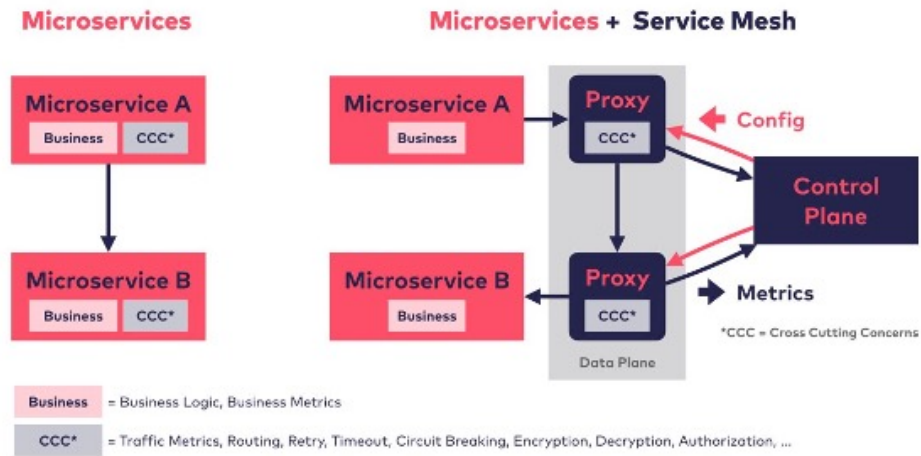


Figure 2.2: Service Mesh structure

The Control Plane receives all the configuration resources and then pushes them to the Data Plane. The first is made by the core components of the mesh while the latter is made by the collection of proxies attached to every microservice. As said before, it allows to abstract the so called *Cross Cutting Concerns* from the microservices in order to put them inside the proxy, so that it'll create a common base for all the actual microservices and the future ones.

Every incoming or outgoing communication is intercepted by the proxy, which then forwards it to the proper destination. In a Kubernetes environment the proxy is another container, known as *sidecar*, inside the pod that contains the microservice.

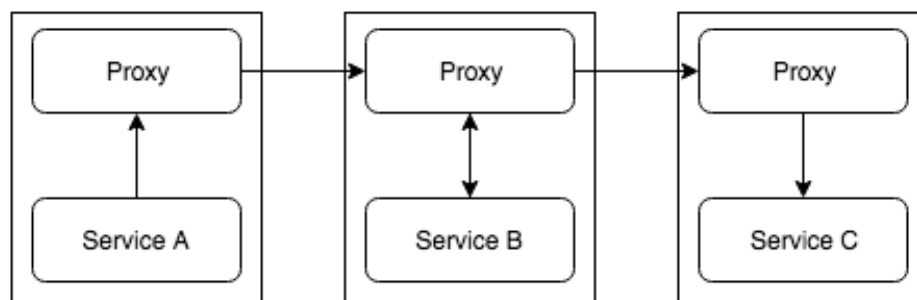


Figure 2.3: Proxies communication example

There are three main categories of features offered by a Service Mesh:

- *Traffic control*: Altogether, the proxies know every connection happening in the mesh and has the power to control it. This allows to apply routing inside the mesh, circuit breaking, quotas and other features depending on the Service Mesh implementation that is been used.
- *Observability*: Service mesh can provide data about the health and the behaviour of the services, collecting and aggregating telemetry data such as latency, distributed tracing, logs and traffic. All of this data gets collected by the Control Plane and can be visualized by integrated tools like Prometheus, Elasticsearch, Grafana, Jaeger and Kiali.
- *Security*: It can automatically encrypt communications inside the mesh using *mTLS*, guaranteeing also mutual authentication. Thanks to this, authorization policies can also be defined. These policies are centrally managed, by the Control Plane, and are applied to service to service communications and to end-users requests. These security features allows to implement the **Zero-Trust paradigm**, which will be discussed later.

2.4 Istio

Istio is a open source implementation of the Service Mesh, supported by the major actors in the cloud. Is the first and most widely used open source project and offers all the advantages of a classic Service Mesh plus some other features like an *Ingress Gateway*.

It is made by a Control Plane, to define and implement the way microservices communicate with each other, and a Data Plane composed by *Envoy* proxies deployed as sidecars in each microservice's pod.

Its Control Plane is composed by a single component called *istiod* which gathers several other components: Pilot, Citadel and Gallery.

Pilot transforms the Control Plane's rules for traffic behavior into configurations that Envoy will use.

Citadel is the center of security; it manages identity management and authentication between services.

User-specified configurations for Istio are taken by Gallery and transformed into legitimate configurations for the other control plane elements.

Being a Service Mesh, it offers a lot of benefits. In this work we'll focus on the security ones. First of all, it offers an automatic implementation of **mTLS** between microservices. It stands for *Mutual Transport Layer Security* and it is a process that establishes an encrypted TLS connection in which both parties use

X.509 digital certificates to authenticate each other. This can be done by automatic issuing to proxies certificates signed by Istio's CA, built in Citadel, which are also automatically rotated. Thanks to mTLS, connections are encrypted and the services are mutually authenticated, this can be used to enforce authorization policies **centrally managed** by the Control Plane. This will be a key point in building a **Zero Trust Architecture** as we will see.

In figure 2.4 we can see an example of the concepts just explained. We can also see the Ingress and the Egress gateways, which are components that manage incoming and outgoing traffic into and from the mesh, respectively. They also implement security features using JWT to authenticate and authorize end-users making requests to the mesh.

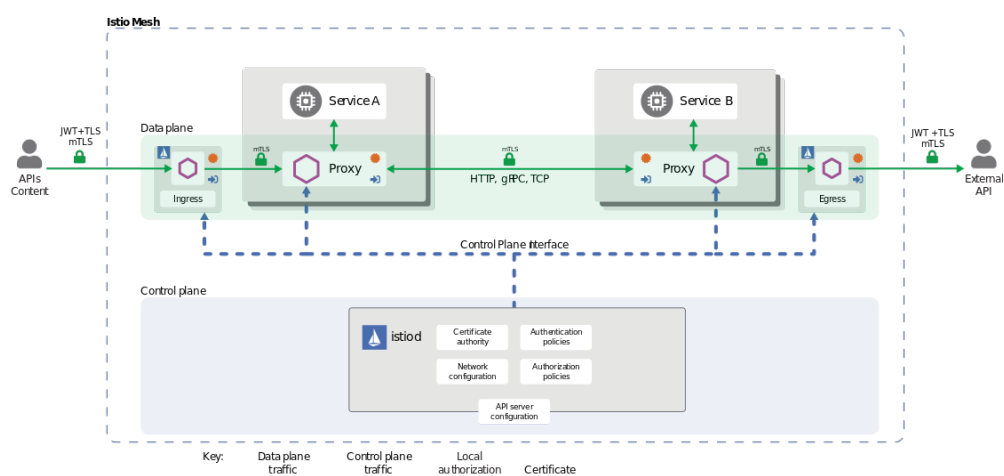


Figure 2.4: Istio Security Architecture example

Security features, as every feature in Istio, are configured using YAML files to deploy the proper resources. There are three main resources to configure those features: Authorization Policy, Peer Authentication and Request Authentication. The first one handles the authorization policies for both service to service communications and end-users requests, while the other two handle authentication policies for, respectively, service to service communications, using certificates assigned to every service, and end users requests, using JWTs.

The most important resources, and the ones that are mostly used in this thesis, are then:

- The Authorization Policy resource
- The Peer Authentication resource
- The Request Authentication resource

- The Virtual Service resource

Since I'm going to cite them in the next chapters, I'll explain how each of them work.

The **Authorization Policy** resource gives the possibility to choose for every microservice a set of rules and the action to perform when at least one rule is matched. The rules match requests from a list of sources that perform a list of operations subject to a list of conditions and a match occurs when at least one source, one operation and all conditions matches the request.

The main fields of the rules are:

- *A set of sources*: they can be microservices, end users, namespaces or IPs.
- *A set of operations*: they can be a list of HTTP methods, a list of paths, a list of hosts or a mix of them.
- *A set of conditions*: they may be mainly a list of headers, JWT claims or a mix of them.

While the action can mainly be:

- *ALLOW*: the request is allowed into the microservice.
- *DENY*: the request is denied to go into the microservice.

Here's an example to clarify these concepts:

The **Peer Authentication** resource defines how traffic will be tunneled (or not) to the sidecar. It's very basic and it's mainly used to apply the STRICT mode to the mesh, a namespace or a single workload. STRICT mode means that all the traffic that interests that particular scope must be mTLS traffic, clear text traffic is not allowed. By default all the mesh is in PERMISSIVE mode, which allows also the clear text traffic, and it's convenient in case of legacy services that aren't inside the mesh and don't have the sidecar yet.

```
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: default
  namespace: foo
spec:
  mTLS:
    mode: STRICT
```

Figure 2.5: Peer Authentication example

In the example above, the STRICT mode is applied to the namespace *foo*, this means that all the inbound and outbound traffic of the services inside that namespace must use mTLS. More than one of this resource can be concatenated in order to obtain more complex situations.

The **Request Authentication** resource defines the rules for the requests that have a JWT, coming from the end-user outside of the mesh. It is usually applied to the Ingress Gateway, since it's the first component that get the requests and it can stop the unauthorized requests as early as possible. The rules are basically based on the *issuer* and on the signature of the JWT. In order to verify the signature a JWKS url can be configured. Note that this is only a first authentication made on the JWT if the request has one, after this the Authorization Policies step in and really decide what to do with that.

Here's an example of a common configuration:

```
apiVersion: security.istio.io/v1beta1
kind: RequestAuthentication
metadata:
  name: jwt-on-ingress
  namespace: istio-system
spec:
  selector:
    matchLabels:
      app: istio-ingressgateway
  jwtRules:
  - issuer: "example.com"
    jwksUri: https://example.com/.well-known/jwks.json
```

Figure 2.6: Request Authentication example

The **Virtual Service** resource, when a host is addressed, specifies a set of traffic routing rules that must be followed. Each routing rule outlines the requirements for traffic of a particular protocol that must match. When a match is made, the traffic is routed to one of the identified destination services (or a subset or version of it) listed in the service registry.

It contains the following fields to specify the traffic affected by it:

- *hosts*: The destination hosts to which traffic is being sent. Could be a DNS name with wildcard prefix or an IP address.
- *gateways*: The names of gateways that should apply these routes.
- *http*: An ordered list of route rules for HTTP traffic.

Then the http rule can contain:

- *gateways*: The names of gateways where the rule should be applied.
- *route*: The actual route to apply to the traffic matching the rule

In the figure below, for example, the resource is applied to the Ingress Gateway, so affects all the traffic going to it. For every request there are two routes that it can match: *example-route-1* and *example-route-2*. The first route is taken if the request's uri has one of those two prefixes and it forwards the request to the *example-service*. The second route is taken if the request's uri is exactly that and it forwards the request to the *bar-service*.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: example-route
spec:
  hosts:
  - "*"
  gateways:
  - istio-ingressgateway
  http:
  - name: "example-route-1"
    match:
    - uri:
        prefix: "/foo"
    - uri:
        prefix: "/example"
    rewrite:
      uri: "/newexample"
    route:
    - destination:
        host: example-service
  - name: "example-route-2"
    match:
    - uri:
        exact: "/bar"
    route:
    - destination:
        host: bar-service
```

Figure 2.7: Virtual Service example

This is a brief overview but this resource is quite complex and rich of possible configurations, more information can be found here [1, Virtual Service]

2.5 Zero Trust Model

It's an IT security strategy that operates under the presumption that no network perimeter is secure and that every communication needs to be verified. The idea of "Zero Trust Security" was developed on the premise that implicit trust is always a weakness and that security must be established using the strategy "Never Trust, Always Verify." Zero Trust restricts access to IT resources in its most basic form by utilizing rigorously enforced identity and device verification procedures.

It's based on four principles, summarized in picture 2.8.



Figure 2.8: Zero Trust Principles

In this case we are focused on enforcing this model inside a cluster. All of these principles can be implemented using a Service Mesh, creating in this way a Zero Trust Architecture inside a cluster:

- *Verify always*: Thanks to the application of authentication and authorization policies to service to service communications, every communication will undergo a process of verification.
- *Least privilege and default deny*: These can be easily implemented using authorization policies. We can let communicate the minimum set of services, useful to fulfill the business logic. This is usually inferred by the application architecture, created by someone who knows exactly the communications topology.

- *Full visibility and inspection:* A Service Mesh, thanks to the proxies, allows to create and collect logs about every single HTTP request made inside the mesh.
- *Centralized management:* Every configuration is implemented in the Control Plane, which is the control *center* of the Service Mesh. It then sends the proper configurations to every proxy which enforces them.

Zero Trust model aims to solve a weakness that have evolved as network topologies and usage have changed. Network security used to be described by a perimeter with distinct lines separating the corporate network's "inside" from "outside". This approach granted users and devices inside the network perimeter the full set of privileges.

Nowadays, however, computing devices are widely dispersed thanks to cloud, mobile, edge, and IoT components, which have blurred the lines of demarcation and made it harder to protect the perimeter. This issue is resolved by the Zero Trust paradigm, which operates under the assumption that nothing, inside or outside the perimeter, can be trusted and that every communication must be authenticated. Zero Trust identification is based on logical attributes rather than the IP address, such as virtual machine names.

The work of thesis, as said, is then based on implementing a Zero Trust Architecture in a cluster that contains multiple microservices. This can be done using a Service Mesh, Istio in this case, and we'll see in the next chapters how it was implemented.

Chapter 3

Design of the Proof of Concept

The main idea was to dive into the concept of the Service Mesh and use it to create a Zero Trust Architecture, that is a cluster architecture where microservices follow the Zero Trust paradigm, in order to understand how it can be created and define some guidelines.

After studying the background concepts the thesis work proceeded with the design of the architecture which was done in a few steps:

1. **Creating a cluster:** this had to be a base step since the Service Mesh, and the microservices inside it, has to be deployed in a Kubernetes cluster.
2. **Developing and deploying a sample application:** since a Service Mesh acts on microservices, a sample application had to be deployed in order to apply a Service Mesh to it.
3. **Installing and configuring the Service Mesh:** obviously, this was a mandatory step.
4. **Exposing the application to internet:** in order to test a component of the Service Mesh and its security features.
5. **Installing and configuring an authorization server:** in order to authenticate end-users and give them a JWT, which would be used to access the application through the Service Mesh.
6. **Accessing external services:** in order to test another component of the Service Mesh and its security features.

Now we will go into details of every design choice for the various steps while the implementation details will be described in the next chapter.

3.1 Cluster creation

Early on, the cluster of choice was *Openshift*, an enterprise PaaS Kubernetes platform made by Red Hat, but given the goal of the PoC it would have been an overkill also because of its price. I then decided to use "vanilla" Kubernetes and moved on to compare the various offers by the cloud providers such as Google, Amazon, Microsoft and IBM. They each offer a managed Kubernetes PaaS service, where all of the platform complexity like the installation and the management of the control plane is handled by the provider.

The various offers by the cloud providers are basically the same and differ a little in cost, a comprehensive analysis was done to individuate the most advantageous one.

In the end, the research showed that the setup effort would have been too much for our use case scenario so the choice fell on *Minikube*, a lightweight Kubernetes implementation that creates a VM on your local machine and deploys a simple cluster containing only one node.

In order to allow the PoC to be accessible also by the tutor, an online machine was needed and the cheapest and easiest choice was an *EC2* machine on AWS, provided by *Blue Reply*.

In this way I had an entire online machine where I could install Minikube, thus creating a small but fully functional cluster well suited for tests and for this PoC.

Minikube requires a *driver*, which means a way to deploy it. It can be deployed as VM, a container, or bare-metal. My choice was to deploy it as a container, using **Docker**, which is the best and preferred choice on Linux.

3.2 Sample application development

As said, in order to test a Service Mesh a sample application was needed and it had to be divided into microservices. Since the application had no special needs and had to be a simple web application, no particular framework was needed. Thus, my first and quick choice was to use **Kotlin** and the **Spring Boot framework**, which is widely used in the enterprise environment because of its versatility and ease of use. Furthermore, it's the framework I know best, considering also that I have some experience with it thanks to a couple of projects that I made for a university course.

In order to have a few microservices I thought to a simple portal in which books are inserted with their descriptions and on which reviews can be made. The structure of this application recalls a demo application provided by Istio but my goal was to rewrite it from scratch to be able to have full control on what it does.

The application was then designed to be composed by four microservices:

1. *Product Page*
2. *Reviews*
3. *Details*
4. *Ratings*

Product Page is the main component and it receives all the requests from the users, it then gathers information from Reviews and Details in order to get all the book info. The other services are not exposed to the internet so their endpoints are used only by the other services.

In order to test a specific feature of the Service Mesh, the application has to consider three kind of users: *Admin*, *User*, and non-authenticated users. The authentication and authorization mechanisms will be done by other two components, the Ingress Gateway and Keycloak, discussed later in this chapter.

The endpoints for every service are the following

Product Page

- `/products` [GET] - allows to retrieve *the titles* of all the books available. This can be used by every kind of users.
- `/products/{bookTitle}` [GET] - allows to retrieve *the details and the reviews* of the book with such title. This can be used by every kind of users.
- `/details` [POST] - allows to post *a new book and its details*, passing a JSON representation of the new book. This can be used only by admins.
- `/reviews` [POST] - allows to post *a new review* passing a JSON object containing the book title and the review information, like a text and the number of stars. This can be used only by authenticated users.

Details

- `/details/{bookTitle}` [GET] - allows to retrieve *the details* of a specific book. It's not accessible by the outside, only the Product Page service uses it.
- `/details` [POST] - allows to post *the details* of a new book. It's not accessible by the outside, only the Product Page service uses it.

Reviews

- `/reviews/{bookTitle}` [GET] - allows to retrieve *the review* of a specific book. It's not accessible by the outside, only the Product Page service uses it.
- `/reviews` [POST] - allows to post *a new review* of a book. It's not accessible by the outside, only the Product Page service uses it.

Ratings

- `/ratings/{reviewId}` [GET] - allows to retrieve *the rating* of a specific review. It's not accessible by the outside, only the Reviews service uses it.
- `/ratings` [POST] - allows to post *the rating* of a new review. It's not accessible by the outside, only the Reviews service uses it.

Each microservice will then have to be connected to a database, so the easiest choice was a single **MySQL** instance containing more databases. This choice is called *SQL server consolidation* and should be subject of analysis as it can decrease the level of security. For example, some points to be checked:

- Who has Admin Access? Is it justified?
- Which are SQL logins permissions?
- Which services uses the SQL server?

The points above for this small example application are quite trivial because the MySQL instance is accessed only by me and the microservices. So to answer those questions:

- There's only one user with admin privileges and I'm the only one that knows username and password. It's justified because I manage the whole architecture.
- I'm the only one with admins permissions. The services only have the minimum set of permissions needed to fulfill their job, such as read and write permissions *only* for their own database.
- Only the four services of the sample application use the server. No other service has the credentials.

So, the PoC is made by "layers" and can be visualized like so:

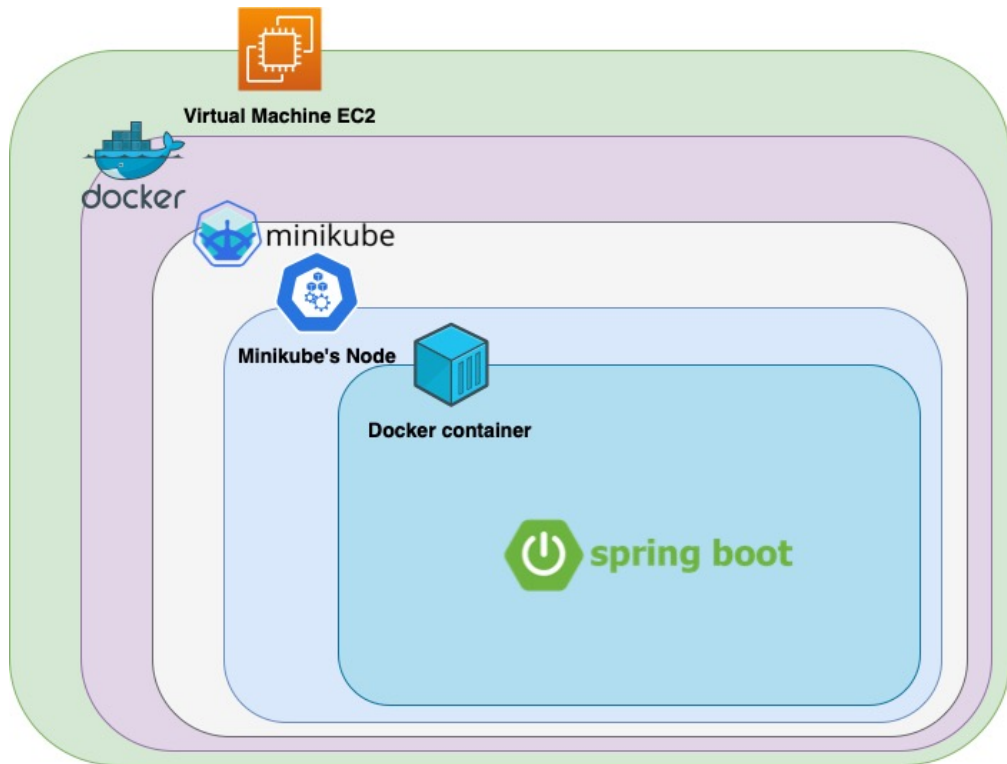


Figure 3.1: PoC Layers

Whilst, the application structure can be schematized as follows:

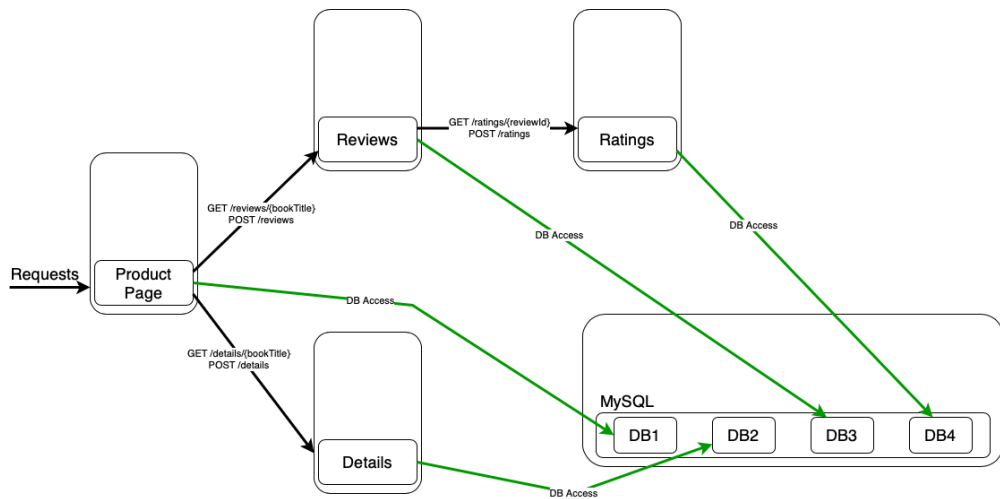


Figure 3.2: BookInfo Application Structure

As said, Product Page is the main component, it gets all the requests coming

from the users and aggregates the information to return. To do that, it asks for the details of a specific book through the endpoint `/details/{bookTitle}` [GET] exposed by the Details service. It also asks to the Reviews service all the reviews of a specific book through the endpoint `/reviews/{bookTitle}` [GET], which in turn asks to the Ratings service the rating of a specific review using the endpoint `/ratings/{reviewId}` [GET]. Then, Product Page composes a JSON containing all these information and returns it to the user.

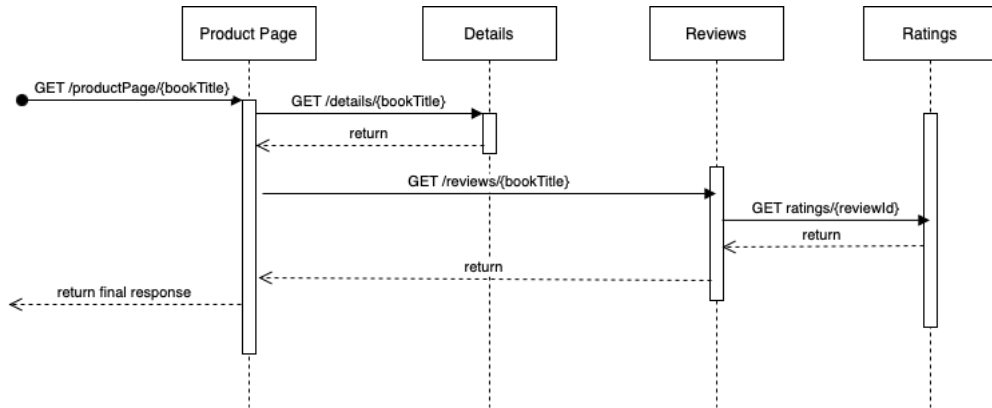


Figure 3.3: Get book details - Sequence diagram

Furthermore, there are two other scenarios: an admin wants to add a new book, a user wants to add a new review. All the checks are **not** done by the application but by the Ingress Gateway.

In the first scenario, the admin sends a JSON representation of a new book to the endpoint `/details` [POST] and the Product Page first saves its title in its own database and then passes the information to the Details service using the endpoint `/details` [POST].

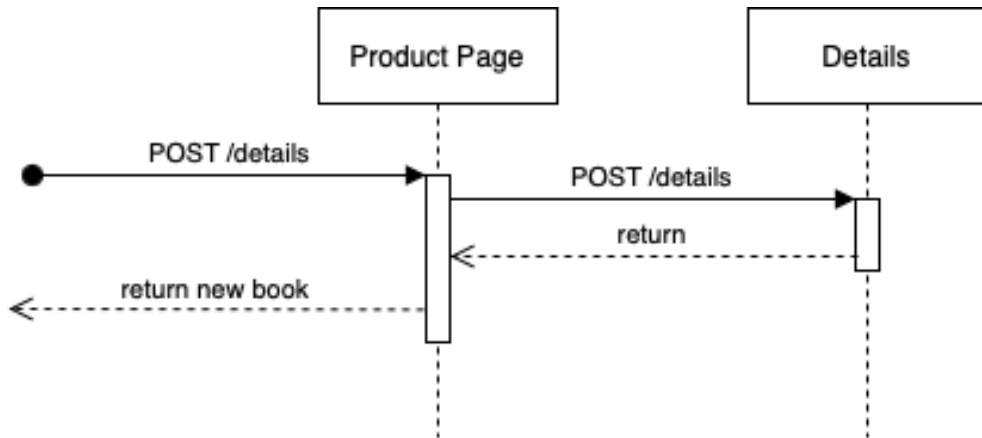


Figure 3.4: Post new book details - Sequence diagram

In the second scenario, the user sends a JSON representation of a new review to the endpoint `/reviews` [POST], the Product Page forwards it to the Reviews service using its endpoint `/reviews` [POST] and then the Reviews service saves all the information except for the rating one which is forwarded to the Rating service along with `review ID` using the endpoint `/ratings` [POST].

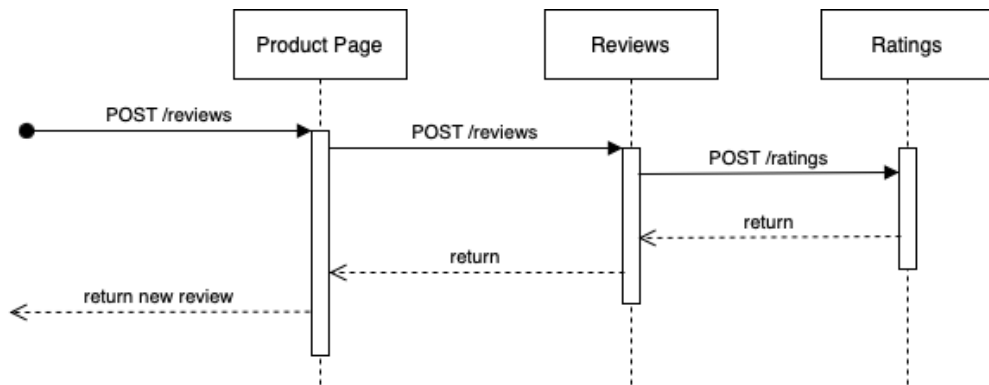


Figure 3.5: Post new review - Sequence diagram




The deployment phase will be described in the implementation chapter.

3.3 Service Mesh installation and configuration

The Service Mesh of choice is **Istio**, which was already described in the background chapter. Istio is the de-facto standard in the Service Mesh world and is used in production by many companies such as Adobe, Baidu, and Google. As an

opensource project is backed up by a large community. Its competitors are *Linkerd* and *Consul* but, as seen in this comparison [2, Service Mesh comparison], it has the most features and flexibility of any of the other Service Mesh by far. This makes it the perfect choice.

Here's a table that compares the three products [2, Service Mesh comparison]:

	 Istio	 LINKERD Linkerd v2	 Consul Consul
Supported Workloads			
	Does it support both VMs-based applications and Kubernetes?		
Workloads	Kubernetes + VMs	Kubernetes only	Kubernetes + VMs
Architecture			
	The solution's architecture has implications on operation overhead.		
Single point of failure	No – uses sidecar per pod	No	No. But added complexity managing HA due to having to install the Consul server and its quorum operations, etc., vs. using the native K8s master primitives.
Sidecar Proxy	Yes (Envoy)	Yes	Yes (Envoy)
Per-node agent	No	No	Yes
Secure Communication			
	All services support mutual TLS encryption (mTLS), and native certificate management so that you can rotate certificates or revoke them if they are compromised.		
mTLS	Yes	Yes	Yes
Certificate Management	Yes	Yes	Yes
Authentication and Authorization	Yes	Yes	Yes
Communication Protocols			
TCP	Yes	Yes	Yes
HTTP/1.x	Yes	Yes	Yes
HTTP/2	Yes	Yes	Yes
gRPC	Yes	Yes	Yes

Traffic Management			
Blue/Green Deployments	Yes	Yes	Yes
Circuit Breaking	Yes	No	Yes
Fault Injection	Yes	Yes	Yes
Rate Limiting	Yes	No	Yes
Chaos Monkey-style Testing	Traffic management features allow you to introduce delays or failures to some of the requests in order to improve the resiliency of your system and harden your operations		
Testing	Yes- you can configure services to delay or outright fail a certain percentage of requests	Limited	No
Observability			
In order to identify and troubleshoot incidents, you need distributed monitoring and tracing.			
Monitoring	Yes, with Prometheus	Yes, with Prometheus	Yes, with Prometheus
Distributed Tracing	Yes	Some	Yes
Multicluster Support			
Multicluster	Yes	No	Yes
Installation			
Deployment	Install via Helm and Operator	Helm	Helm
Operations Complexity	How difficult is it to install, configure and operate		
Complexity	High	Low	Medium

After creating the cluster and deploying the application it was the turn of Istio.

The installation phase was straightforward and didn't require any design process while the configuration phase required some thought.

This paragraph exposes the design of the "internal" mesh configuration, while the configuration for the incoming traffic using the **Ingress Gateway** and the outgoing traffic using the **Egress Gateway** will be explored later.

As we said earlier, the four main points to achieve the Zero Trust paradigm are: *Verify always*, *Least privilege and default deny*, *Full visibility* and *Centralized management*. So the goal is to configure Istio in order to fulfill Zero Trust paradigm characteristics.

Istio control plane acts through **centralized management** and the proxies **always verify** the authentication policies before service to service communications, so in the configuration phase I only had to take care of "least privilege and default deny" principle and TLS configuration.

TLS is enabled by default in Istio in the so called *PERMISSIVE* mode, which means that a workload accepts both TLS and plain traffic. Therefore the only step needed was to switch to *STRICT* mode so that workloads only accept TLS traffic. Furthermore, Istio implements **mutual** TLS, so microservices are mutually authenticated, which is another key point in the Zero Trust paradigm. This can be done using the Peer Authentication resource.

On the other hand, **least privilege and default deny** paradigm needs to be implemented through the Authorization Policy resource, enforced by the proxies. An explanation about how these two resources work can be found in the paragraph about Istio in chapter 2.

It can be used to enforce the schema proposed in the figure 3.2, so for example the Details microservice can only be contacted by the Product Page microservice and only using the GET and POST verbs.

```
apiVersion: "security.istio.io/v1beta1"
kind: "AuthorizationPolicy"
metadata:
  name: "productpage-to-details"
  namespace: default
spec:
  selector:
    matchLabels:
      app: details
  rules:
  - from:
    - source:
      principals: ["cluster.local/ns/default/sa/bookinfo-productpage"]
    to:
    - operation:
      methods: ["GET", "POST"]
```

Figure 3.6: Details service Authorization Policy

In this case this Authorization Policy is applied to the Details microservice and I only need to specify one source, the Product Page service, specified as a *principal* which means the service name in Istio conventions, and the methods allowed. The action is ALLOW and it's omitted because it is the default one. This is because if a workload has *at least one* Authorization Policy then if any request coming to that workload doesn't match any of them then it's automatically denied. This applies the **default deny** principle.

As we will see in the implementation chapter, where we'll go deeper into the code, the other Authorization Policies will be very similar to this one because the goal is mostly the same.

It's important to note that the **least privilege** property is totally left to the operator that configures Istio. He's the one who must translate the communication topology into the proper Authorization Policies, ensuring that every service can talk only to the services that it needs to contact. If he makes any mistake then it would leave open routes that do not strictly serve the functioning of the application, increasing the attack area.

After Istio installation the application structure can be updated like so:

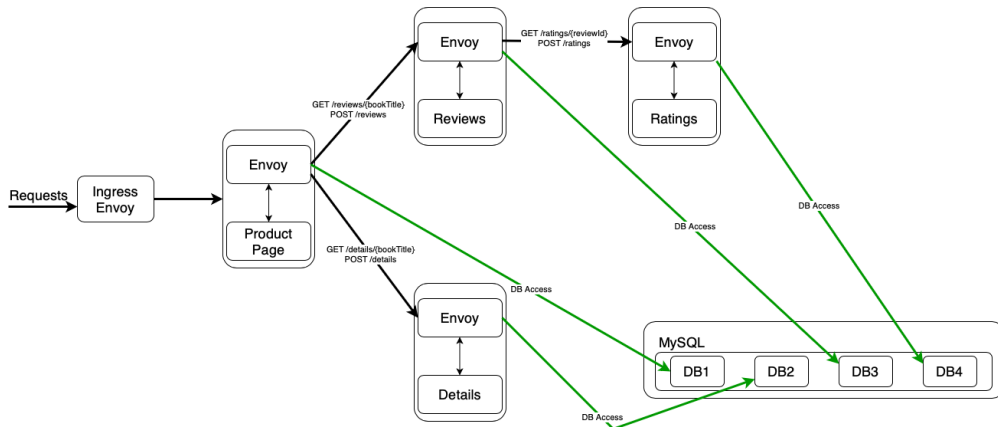


Figure 3.7: BookInfo structure using Istio

We can now notice the **Envoy proxies**. As said, these are the proxies that Istio uses to enforce its configuration and they form the *data plane*. Editing the IP tables of the microservices they can intercept all the traffic outgoing from the service as well as the incoming one, applying rules or security policies like the authorization ones we have just seen. In practice, from now on the services won't talk directly anymore but only through their proxies, which are capable of enforcing all the configurations that we send to Istio's control plane.

3.4 Application exposition

Once the application and Istio are setup, the next step was to make the whole infrastructure reachable by the outside. This requires two components:

- A software **load balancer** since Minikube doesn't have one. This is needed to expose IPs outside the cluster and is done by **MetalLB**.
- A component that makes the application accessible from outside the cluster **but inside the EC2 machine**. This is done by the **Ingress Gateway**.
- A component that is exposed **outside of the EC2 machine** and forwards the requests to the Ingress Gateway. This is done by a **reverse proxy**.

MetalLB is a software load balancer that assign *Load Balancer* type IPs to any service that needs one. This is the case of the Ingress Gateway, since it needs to be reachable from the outside.

The *Ingress Gateway* is basically a service offered by Istio that gets exposed, receives all the requests and then routes them inside the mesh. In this way there's

only one entry point, allowing to implement authentication and authorization controls thus improving the security.

After its "activation" the only route we need is the one to the *Product Page* microservice, so every request only needs to be authorized, authenticated and then routed directly to it. The authentication and authorization are based on information that will be described later, when the authorization server will be presented.

The Gateway obviously cannot be reached outside the EC2 machine, this is the job of the *reverse proxy*. The most common reverse proxy is **Nginx**, it listens to a port and forwards every request to the local Gateway. The request in this case is forwarded as is, in order to preserve especially the headers which contain, among the other things, the JWT.

It's important to notice that for security reasons I set the firewall of the EC2 machine to only allow Reply's VPN IPs.

Ultimately, we now have a fully functional application, inside an Istio Service Mesh, that can be reached from outside the EC2 machine. The architecture is now in this state:

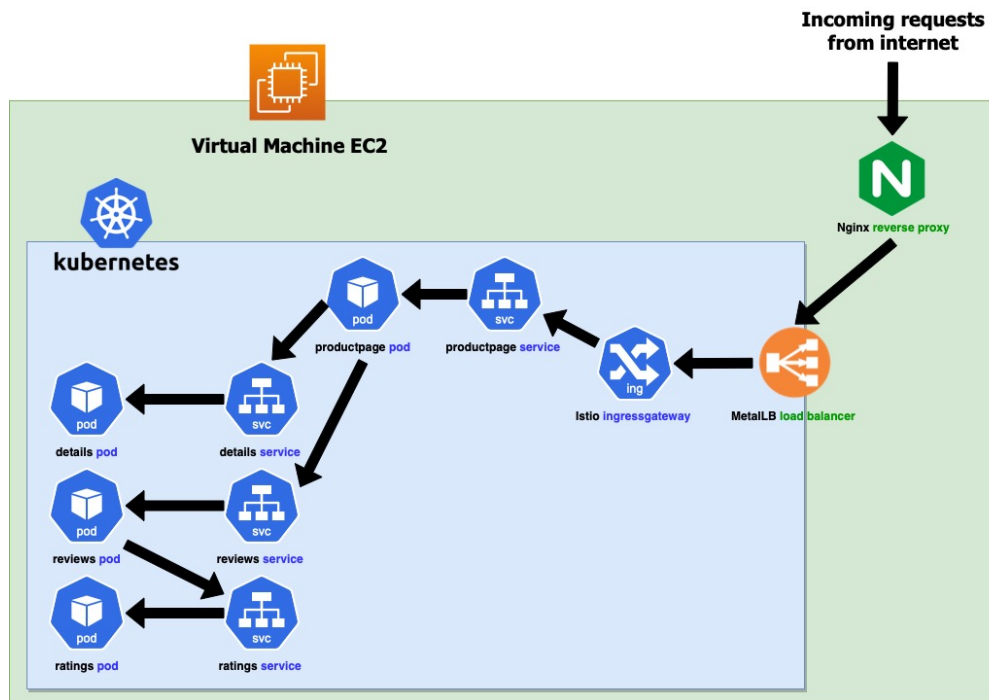


Figure 3.8: PoC Intermediate Structure

3.5 Authorization server installation and configuration

The authorization server (Identity and Access Management) is needed to create users and display endpoints through which users could receive a **JWT access token** to be used later to authenticate with the application and test the Ingress Gateway security features.

I used **Keycloak** which is an open source software product that enables **Single Sign-On (SSO)** with Identity Management and Access Management for modern applications and services. It is written in Java and supports identity federation protocols such as **SAML v2** and **OpenID Connect (OIDC) / OAuth2**. It is licensed by Apache and supported by Red Hat.

Its main competitors are *Okta Single Sign-On* and *PingOne Cloud Platform*. They are all cloud native and offer similar features but Keycloak stands out because it's opensource, which also means free, has a lot of documentation and it's easier to install and configure. These characteristics made me choose Keycloak. Despite all of its powerful features, my goal was only to create some users with different roles and give them a way to authenticate and get a JWT. This is done by all the three products cited above but Keycloak, after looking through the guides of each product, seemed to be very easy to configure as we'll see in the next chapter. It is also one the most popular cloud native IAM right now, being supported by Red Hat, and it's been used in many production environments, as Blue Reply confirmed based on its clients. My tutor was also very expert in Keycloak, more than the other products, so his support also played a significant role in the choice.

I needed to deploy it in the cluster in order to be accessible by the users and I needed to connect it to an external database to simulate a production environment. Fortunately, Keycloak is well suited for this kind of deployment and has a prebuilt image that can be easily deployed in a cluster. After that you just need to configure the DB and add the users.

I chose to deploy it in a separate namespace for the *separation of concerns* principle and also because I don't think that is a good idea to place it inside the mesh if we wanted to simulate a real environment. In a production environment the authorization server could be deployed in a completely different place respect to the application and handled by another company, so it couldn't be included in the mesh even if we wanted to.

It's important to note that Keycloak doesn't adhere to the Zero Trust principles because it is a separate component from the application. Being an authentication server it allows already registered users to authenticate and it responds to them with a JWT, used later to contact the application. Keycloak does not communicate with any other component but only with users, so it has no one to "verify first".

Theoretically, it may be handled by a completely different entity and it may reside on unknown servers, and this is very common in the modern internet. Since only the users must communicate with it, there's no need to include it in the Zero Trust architecture. The only verification that the application does is on the signature of the JWT that it issues and is being used by the users, and this is enough.

Once installed, deployed and attached to a database inside the MySQL server, the configuration consisted in configuring the *client*, which is the application, adding the roles within the client, which are *Admin* and *User*, and creating some users with a certain username, password and role.

Finally, Keycloak exposes various endpoints, the ones that we need are the *access token* and the *JWKS* endpoints:

- `http://<base_url>/realms/<realm>/protocol/<protocol>/token`
- `http://<base_url>/realms/<realm>/protocol/<protocol>/certs`

The first one receives various parameters in the body of the request:

- *client_id*: a unique client ID that identifies a single client inside a realm, it is set during the client configuration phase.
- *grant_type*: the type of **Access Grant Flow** for authenticate the user. In this simple case the **password** type will always be used.
- *scope*: the protocol used to verify the identity of the user. In this case it will always be **openid** that stands for *OpenID Connect*.
- *username*: user's **username**, required if the *grant_type* is *password*.
- *password*: user's **password**, required if the *grant_type* is *password*.

It responds, among the other things, with an access token that can be then used to authenticate to the application.

The second one corresponds to the **JWKS uri**, which is the location of the JWKS, a set of keys containing the public keys used to verify any JSON Web Token (JWT) issued by the authorization server. This one is used by the *Ingress Gateway* during its checks.

The architecture is now improved with this new component:

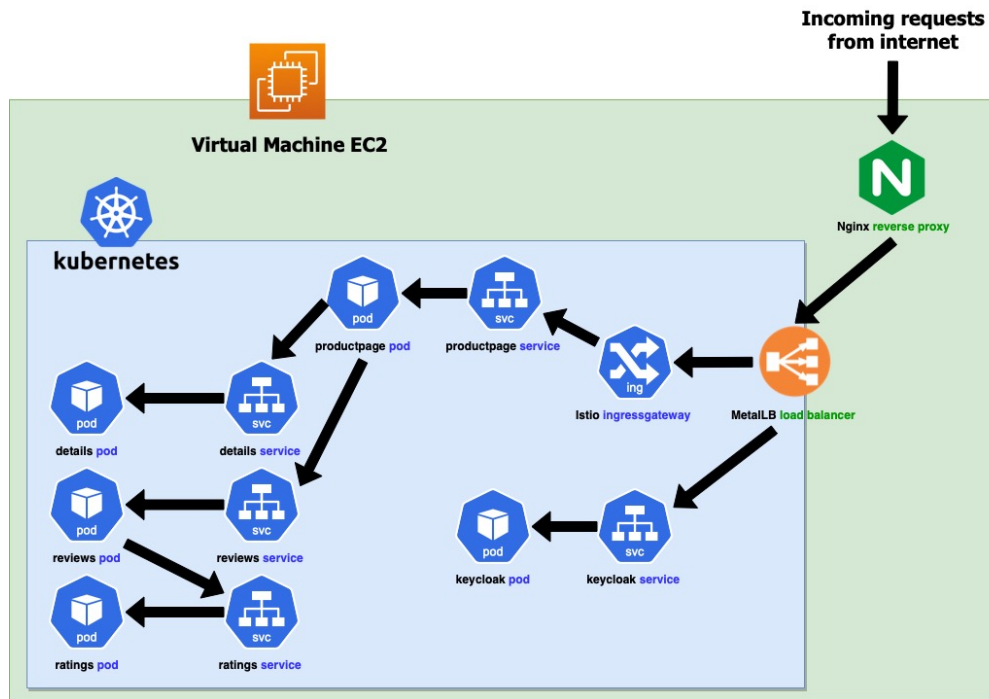


Figure 3.9: PoC with Keycloak

3.6 External services access

The last Istio's main component to use and test is the **Egress Gateway**. It allows microservices to reach external services using a dedicated and centralized component.

This allows to apply Istio features, for example, monitoring and route rules, to traffic exiting the mesh.

For example, take into account a company with a stringent security policy dictating that any traffic leaving the service mesh must pass via a specific set of dedicated nodes. These nodes will operate independently from the cluster's other nodes that are running applications on dedicated machines. These unique nodes will be more closely watched than other nodes in order to enforce the policy on egress traffic and each of them will implement the Egress Gateway.

Another use case, which is very similar to the one of my PoC, is a cluster where the in-mesh services cannot connect to the Internet because the application nodes do not have public IP addresses. The application nodes are able to access external services in a regulated manner by defining an Egress Gateway, routing all egress traffic through it, and assigning public IP addresses to the egress gateway nodes.

In my case in-mesh services *may* have public IP addresses but, for security

reasons, in order to regulate the access to external services, they are forced to pass through the gateway.

In order to use the Egress Gateway we need to route all the traffic going outside the mesh to it, this is done creating routing rules through the *Virtual Service* Istio's resource. Then, from the Gateway we need another routing rule in order to send that traffic to the right external service.

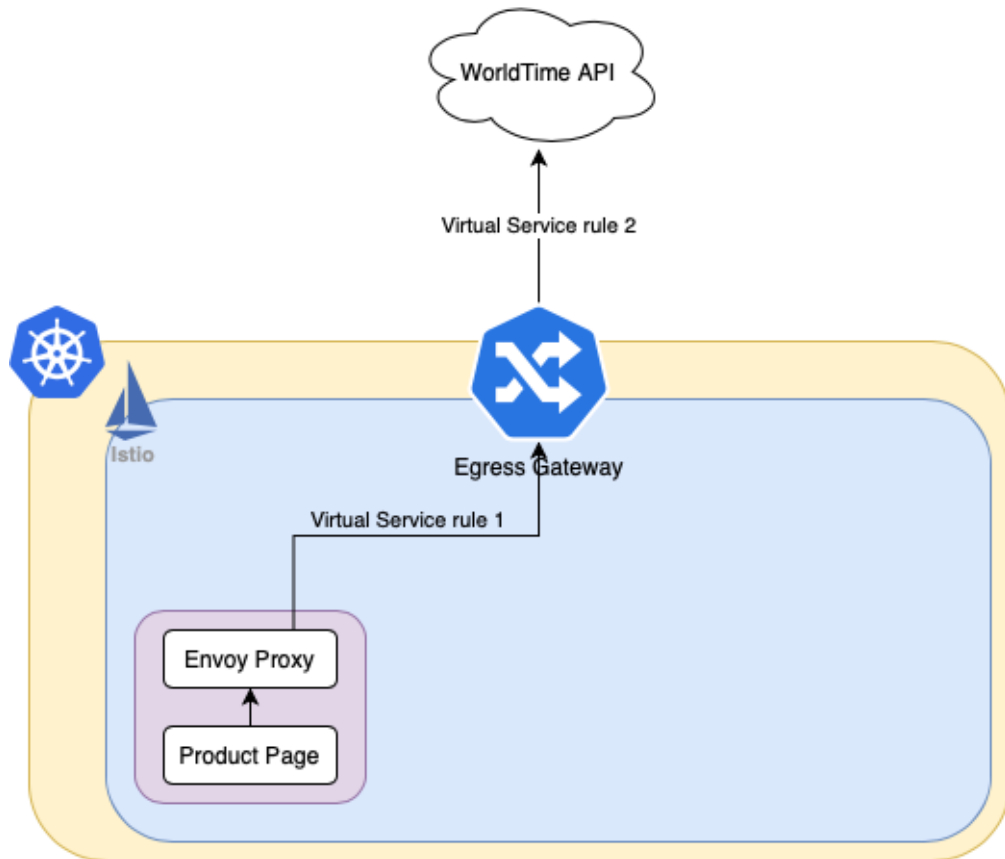


Figure 3.10: Egress Gateway example

An explanation of this resource can be found in the paragraph about Istio in chapter 2, instead the code of the *Virtual Service* for contacting the external service through the Egress Gateway will be in the implementation chapter.

Istio can also be configured to automatically deny connection to services external to the mesh, thus reducing the attack surface if a pod is compromised. An attacker couldn't contact an arbitrary service under his control to deal more damages.

Now another great feature comes in handy, the **Service Entries**. They allow to include extra entries to Istio's **internal service registry**, permitting access and routing to them, from services already in the mesh. A *Service Entry* lists a

service's attributes, including its DNS name, VIPs, ports, protocols, and endpoints. These services may not be internal to the mesh and that's the main point because Istio can deny connection to external services *unless* they are inside its internal registry, that is they are inserted as Service Entries.

In order to test all of the above, I needed to block the connection to unknown external services through Istio configuration and add a request to an external service from the sample application. The call is a bit useless and it's made by *Product Page* to a timestamp service, called `worldtimeapi.com`, which returns the current local time for a given timezone as JSON. The microservice just gets the timestamp JSON and places it in the response to the client, only for testing purposes.

To allow this, that service must be inserted as a Service Entry so that Istio won't block the call.

It's important to note that the Egress Gateway is a "secondary" component compared to the rest. It was inserted to test all of Istio's components but it has not been used to provide mutual authentication with an external service. Configuring an external service that supports mTLS is not trivial since it should be done by hand. The connection is not automatically established by Istio, and would require two certificates, one for the Egress Gateway and one for the external service, so this has not been investigated.

Finally, with this last piece of the puzzle the architecture is done and can be summarized like this:

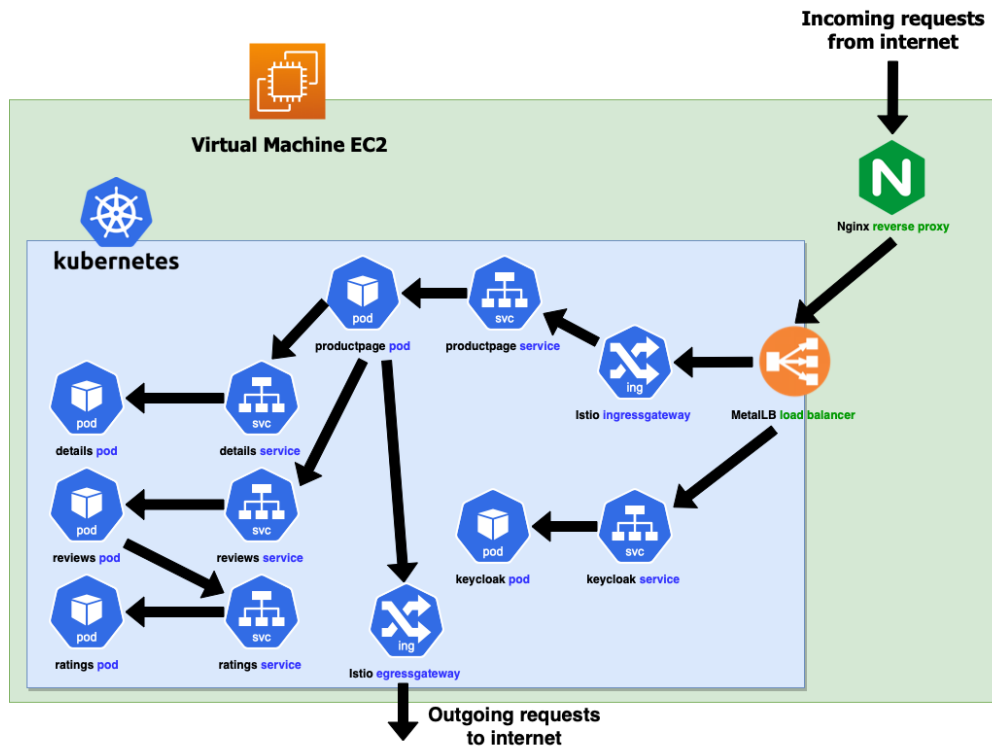


Figure 3.11: PoC Final Structure

In the chapter 5 the verification of the achievement of Zero Trust will be illustrated.

Chapter 4

Implementation of the Proof of Concept

The goal we wanted to achieve was to install an **Istio** service mesh in a **Kubernetes** cluster in order to implement the **Zero Trust** ideology among the microservices within the cluster. Briefly, this was implemented on an **EC2** machine on *AWS*, using **Minikube** to create a cluster locally with one master and one worker node. Once the cluster was created, a simple **Spring Boot** application called *BookInfo* was created. This application works using a **MariaDB** database installed directly on the machine. The service mesh was then installed via the Istio CLI. Subsequently, **MetalLB**, a software load balancer that can be installed as a Minikube addon, was installed in order to assign external IP addresses to the cluster services. To then make the cluster available outside the EC2 machine, **Nginx** was installed on the machine and used as a *reverse proxy*. The last step was the deployment of **Keycloak** on the cluster. It is an open-source IAM and is used to provide JWT tokens in order to interact with the application deployed on the cluster. In the following paragraphs, each one of these steps will be described in more detail.

4.1 EC2 machine creation

The machine was created on *AWS* using an *XLarge* format instance with Ubuntu 20 operating system. Once created, it was possible to connect to the machine via **SSH** using the default user *ubuntu*. As a good practice, I subsequently created the user *andreascopp* and, to make my life easier, I enabled access via an asymmetric challenge using a private key. A security group was also created, that is firewall rules, that allows access to the machine only through the IP of Reply's VPN using port 22, i.e. **SSH**.

4.2 Minikube installation

Minikube was installed using `apt`, following an online guide [3, Minikube Guide]. Then Kubectl and Docker were installed. In particular, Minikube configuration assigns 8 GB of RAM and 4 vCPUs to the cluster, necessary to make Istio work properly. Furthermore, Minikube is started using Docker as **Driver** and in this way a container is created, which contains Minikube and therefore the whole cluster.

4.3 Istio inside Minikube installation

Istio was installed following an online guide [4, Istio Guide]. It was then installed by downloading the CLI and using the `install` option. This allows you to install Istio using one of the profiles provided, in this case I used the `demo` profile which provides the components **istiod** (the core of Istio), **Ingress Gateway** and **Egress Gateway**.

I also installed all the addons made available by Istio in the form of YAML files in one of the folders of Istio's source files. These addons can be used for statistics or to have other useful information about the status of the mesh. For example, Kiali is one of the main addons and it provides a dashboard that gives you an overview of the mesh, showing the service graph, the health of the services and the health of the connections between them. It also provides a useful tool that detect the basic errors in Istio configuration files.

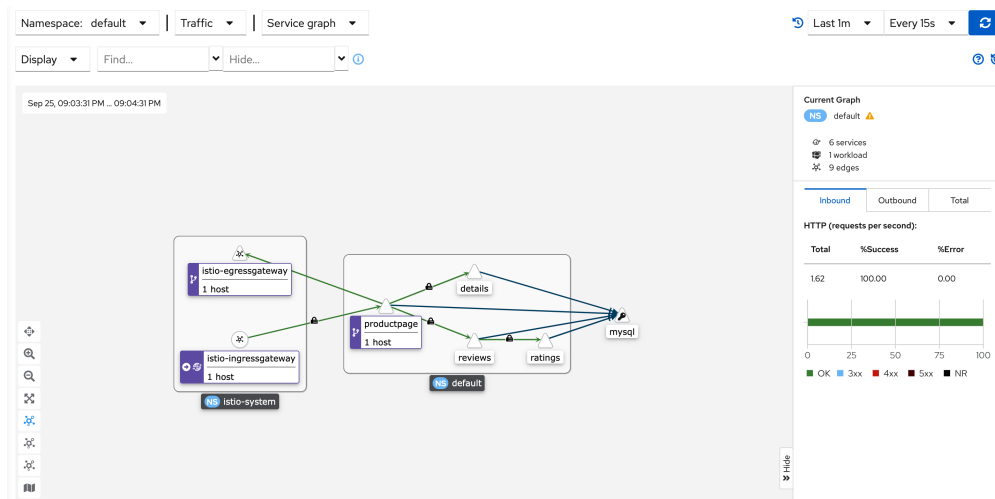


Figure 4.1: Kiali dashboard

After the installation, a new namespace called *Istio-system* is created, within which all the Istio components are installed. Since by default the injection of

the **Envoy proxy** is automatic, in each namespace where you want to perform the injection and then install the mesh, just apply an `istio-injection=enabled` label.

It must be specified that due to incompatibility problems with MariaDB [5, Server First Protocols], all calls made to the DB are not intercepted from proxies but are done directly by microservices. This was done by adding the `-set values.global.proxy.includeIPRanges = "10.96.0.0/12"` flag during the install command. This specifies the proxy to intercept only communications in that IP range, which is the Minikube range. Various resources made available by Istio were then used and they will be discussed later.

4.4 Development and deployment of BookInfo application

The application is the heart of the PoC because it allows to experiment with all the other tools. As said, it is an application written in **Kotlin** using **Spring Boot** and represents a portal in which books are inserted with their descriptions and on which reviews can be made.

I used IntelliJ IDEA which has a lot of nice features for the Spring Boot framework, as well as for Kotlin since JetBrains, IntelliJ's developer, is the company that developed it.

For each project, I initialized it using *Spring Initializr* inside IntelliJ adding the library *Spring Boot Starter Data JPA* and the MySQL driver. Then, I created the following directories to organize the classes of the source code:

- *controllers*: contains the controllers classes.
- *domain*: contains the entities.
- *dto*: contains the DTOs (Data Transfer Objects).
- *exceptions*: contains the custom exceptions, useful for each service.
- *repositories*: contains the repositories to communicate to the DB.
- *services*: contains the services, the core of the business logic.

Each class is quite simple since the application doesn't do a lot of work, it only serves the purpose of testing Istio. For example, here's the controller of the Product Page service for the `/products` endpoints:

```
@RestController
@RequestMapping("/products")
class ProductPageController(
    val productPageService: ProductPageService,
    val timestampRestService: TimestampRestService,
    val requestService: RequestService
) {
    private val logger: Logger = LoggerFactory.getLogger(ProductPageController::class.java)

    @GetMapping
    fun getAllProducts(): ResponseEntity<List<String>>{
        return ResponseEntity(productPageService.getAllTitles(), HttpStatus.OK)
    }

    @GetMapping("/{bookTitle}")
    fun getProduct(
        request: HttpServletRequest,

        @PathVariable
        bookTitle: String
    ): ResponseEntity<ResponseDTO>{
        logger.debug("Book title: $bookTitle")

        val normalizedBookTitle = bookTitle.replace( oldValue: "%20", newValue: " ")
        logger.debug("Normalized book title (without %20): $normalizedBookTitle")

        val bookDTO = productPageService.getBookInfo(normalizedBookTitle)
        val clientIp = requestService.getClientIp(request)!!
        val timestampDTO = timestampRestService.getTimestampByIp(clientIp)

        return ResponseEntity(ResponseDTO(bookDTO, timestampDTO), HttpStatus.OK)
    }
}
```

Figure 4.2: Product Page products controller

For the application's deployment, I wrote four **Dockerfiles** to create four **Docker images**, one for each service, which were then uploaded to **Docker Hub**. Each Dockerfile simply starts from an *OpenJDK* imagine, copy the jar of the service inside the image and then starts it. Here's the Product Page Dockerfile but the other ones are the same, it only changes the name of the jar file:

```
FROM openjdk:11-jdk-oracle

ARG JAR_FILE=build/libs/BookInfo-ProductPage-0.0.1-SNAPSHOT.jar
COPY ${JAR_FILE} app.jar

RUN mkdir -p ./tmp

EXPOSE 8080

ENTRYPOINT {"java", "-jar", "./app.jar"}
```

Figure 4.3: Product Page Dockerfile

Next, four YAML file was created. Each describes three resources for needed by the microservices: *Deployment*, *Service*, and *Service Account*.

The **Deployment** indicates how the pod should be structured, then specifies, among other things, the image to use, the number of replicas and the ports exposed.

The **Service** indicates how that Deployment should be exposed. In this case, they are all exposed only within the cluster via *ClusterIP* so that the services can only be contacted among themselves.

The **Service Account** is used to identify the services and therefore to be able to specify authorization policies when the services talk to each other.

Here's the Details service YAML file. The other ones are similar but the structure is almost the same since they need the same resources:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: details-v1
  labels:
    app: details
    version: v1
spec:
  replicas: 1
  selector:
    matchLabels:
      app: details
      version: v1
  template:
    metadata:
      labels:
        app: details
        version: v1
    spec:
      serviceAccountName: bookinfo-details
      containers:
      - name: details
        image: docker.io/andreasco/bookinfo-details:latest
        imagePullPolicy: Always
        ports:
        - containerPort: 8082
        securityContext:
          runAsUser: 1000
```

Figure 4.4: Details Deployment YAML


```
apiVersion: v1
kind: Service
metadata:
  name: details
  labels:
    app: details
    service: details
spec:
  ports:
    - port: 8082
      name: http
  selector:
    app: details
```

Figure 4.5: Details Service YAML

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: bookinfo-details
  labels:
    account: details
```

Figure 4.6: Details Service Account YAML

All these files were then grouped into a single file for convenience, so that the whole application could have been deployed using the command `kubectl apply -f bookinfo.yaml`.

4.5 Ingress Gateway and end-users policies implementation

The **Ingress Gateway** is provided by Istio, is exposed outside the cluster, receives the requests and then do the routing to internal services based on rules. It is deployed within the `istio-system` namespace and is configured with a specific Istio resource called **Gateway** that configures the port where the Gateway "listens" and the protocol used.

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: istio-ingressgateway
spec:
  selector:
    istio: ingressgateway # Use istio default controller
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
      - "*" # Using this every host can use the gateway
```

Figure 4.7: Ingress Gateway YAML

Subsequently, internal routes are created using a resource called **Virtual Service**, in this case the requests are all sent to the *Product Page* service. This rule is specifically for all the traffic that matches the prefix path `/api/v1/`, it rewrites the prefix to `/` and then sends the request to the port 8080 of the *productpage* Kubernetes service

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: bookinfo-vs
spec:
  hosts:
  - "*"
  gateways:
  - istio-ingressgateway
  http:
  - match:
    - uri:
        prefix: /api/v1/
      rewrite:
        uri: "/"
      route:
      - destination:
          host: productpage
          port:
            number: 8080
```

Figure 4.8: Ingress Gateway Virtual Service YAML

The gateway also implements end-users authorization policies via **JWT** using the **Request Authentication** resource. It specifies that all those who contact the Gateway by specifying an Authorization header, which therefore indicates a JWT, must have a JWT issued by **Keycloak**, while for requests without JWT this policy does not apply and will be the authorization policies to to decide what to do.

```
apiVersion: "security.istio.io/v1beta1"
kind: "RequestAuthentication"
metadata:
  name: "jwt-token-request-authn"
  namespace: istio-system
spec:
  selector:
    matchLabels:
      app: istio-ingressgateway #It's applied to the gateway so can check the requests as early as possible
  jwtRules:
  - issuer: "http://keycloak.com/realms/istio"
    jwksUri: "http://keycloak.com/realms/istio/protocol/openid-connect/certs"
```

Figure 4.9: Ingress Gateway Request Authentication YAML

End-user authorization policies are implemented through **Authorization Policies** resources that take the metadata extracted from requests through Request Authentications and use them to take decisions.

In particular, policies have been implemented to give permissions to users as follows:

- *Admins*: can get the books list, add and remove new books.
- *Users*: can get the books list and post new reviews for existing books.
- *Unauthenticated users*: can only get the books list.

This is done by reading the role that has been assigned to the user among the *claims* of the JWT, if there's no JWT it means that the user is not authenticated.

These policies are very similar to the one seen in figure 3.6 but in this case they take decisions based on the JWT claims. For example here's the policy that allows *Admins* to get book details, the other ones are very similar:

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: allow-admin-get-book-info
  namespace: istio-system
spec:
  selector:
    matchLabels:
      app: istio-ingressgateway
  action: ALLOW
  rules:
  - from:
    - source:
      requestPrincipals: ["http://keycloak.com/realms/istio/*"]
    to:
    - operation:
      hosts: ["myapp.com"]
      methods: ["GET"]
      paths: ["/api/v1/products/*"]
    when:
    - key: request.auth.claims[resource_access][bookinfo][roles]
      values: ["ADMIN"]
```

Figure 4.10: Authorization Policy for Admins - GET

4.6 Intra-services policies implementation

The authorization policies for communications between services are implemented using the *Authorization Policies* seen before. Now they use the certificate created by Istio, that contains a name assigned to the service using the Service Account resource of each service. The policies allow to specify for each service which other service can call it and with which *RESTful* verbs.

A resource was therefore created for each service to enforce the communication scheme shown in figure 3.7.

Also, it should be noted that this works because **mTLS** is active between services, implemented by the Envoy proxy. It is active by default in internal communications between microservices and is automatically implemented using certificates issued by **Istiod**, the core of Istio that integrates the old **Citadel** which also acted as **CA** (Certificate Authority). With mTLS, services therefore have a certificate that also specifies their identity and it is then possible for authorization policies to make decisions based on it.

Furthermore, as said in the previous chapter, in this case I switched to the STRICT mode using the **Peer Authentication** resource:

```
apiVersion: "security.istio.io/v1beta1"
kind: "PeerAuthentication"
metadata:
  name: "default" #Mesh-wide policies must be named "default"
  namespace: "istio-system" #And must be deployed in Istio installation namespace
spec:
  mtls:
    mode: STRICT
```

Figure 4.11: Peer Authentication - STRICT mode

4.7 MetalLB configuration

MetalLB is a software **load balancer** that is used in Minikube because, being on bare metal, it does not have a cloud load balancer that is provided for example by cloud providers. It is used to assign IP addresses external to the cluster to the services that require them, in our case Keycloak and the Ingress Gateway.

It was simply installed as a Minikube addon using the command `minikube addons enable metallb`. After the installation it requires an IP range that it will use to assign new IPs to services that require one. In this case, they start from the Minikube IP up to a range of 20 addresses. After this, every service that requires an external IP will have assigned the first one free in that range, in ascending order.

4.8 Nginx configuration

Nginx is used as a *reverse proxy*, which is a proxy that listens on a certain port of the EC2 machine and forwards requests to the IP addresses within the machine based on the path to which the initial request refers to. For example, in our case the proxy listens on port 51999, if an HTTP request arrives with the path `/auth`, the request is sent to the IP address to which Keycloak is exposed, which is a *Load Balancer* so it's external to the cluster but internal to the machine. All the other requests are instead sent to the IP to which the Ingress Gateway is exposed, which then routes them to the application.

```
server {
    listen 51999;

    proxy_set_header X-Forwarded-For $proxy_protocol_addr; # To forward the original client's IP address
    proxy_set_header X-Forwarded-Proto $scheme; # to forward the original protocol (HTTP or HTTPS)
    proxy_set_header Host $host; # to forward the original host requested by the client

    location / {
        proxy_http_version 1.1;
        proxy_pass http://myapp.com;
    }

    location /auth {
        rewrite /auth(.*) /$1 break;
        proxy_http_version 1.1;
        proxy_pass http://keycloak.com;
    }
}
```

Figure 4.12: Reverse Proxy configuration

Nginx was installed directly on the machine via `apt-get` and it is configured via a configuration file located at the path `/etc/nginx/sites-available`.

4.9 Keycloak configuration and deployment

Keycloak can be set through the UI where you can create *Realms*, *Clients*, or applications to be protected, and add users and roles that a client manages. Once the users have been created, they can be assigned a password and one or more roles within the client.

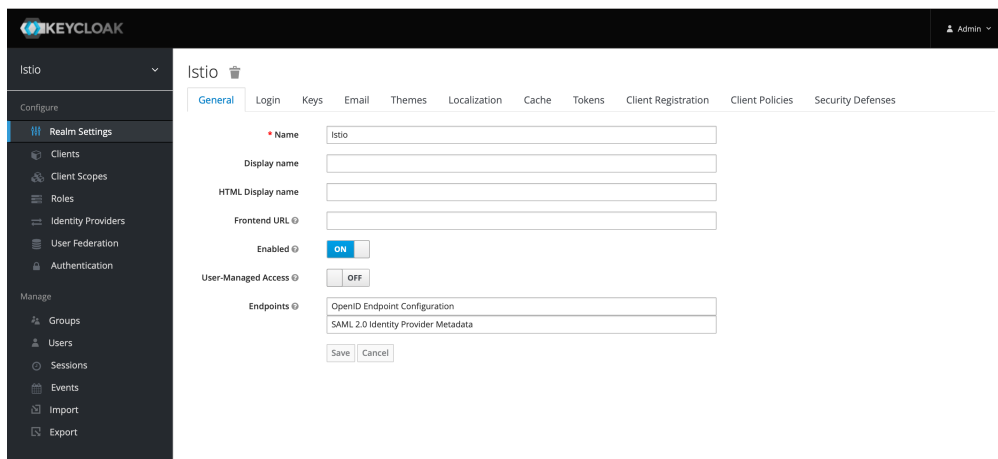


Figure 4.13: Keycloak dashboard

It has been deployed in a specific namespace called STS via a *Deployment* resource and a *Service* resource, and is exposed directly to the outside via an external *Load Balancer* IP set by the Service resource.

Its deployment was easy since a YAML template of the Deployment, along with a base Docker image, was already provided by the developers. My main effort was for the Service resource in order to expose it. After the deployment the basic configuration was easy thanks to a very intuitive dashboard that allowed me to set the roles that the users could have in a couple of clicks.

In order to do that, I created a client, which is the application that Keycloak is serving as authorization server, and I configured that client to have users with the roles ADMIN or USER. After that, I manually created a couple of users composed by username and password. These credentials need to be passed to a specific Keycloak endpoint to obtain a valid JWT:

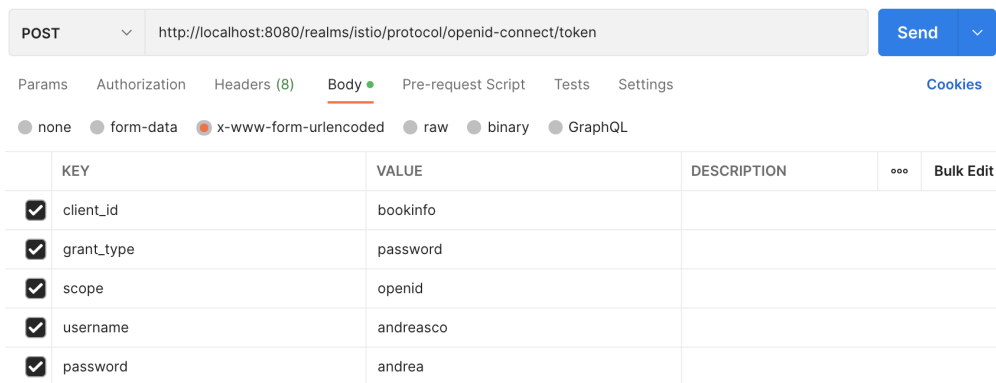


Figure 4.14: Postman REST request to obtain JWT

Where the body is passed as `x-www-form-urlencoded` data.

In the figure 4.15 is shown only the Service part of the YAML because it's important to notice that Keycloak is *directly* exposed outside of the cluster. This is because Keycloak is *not* part of the Service Mesh, since it doesn't need to communicate with other services, so the Gateway cannot route the requests to it.

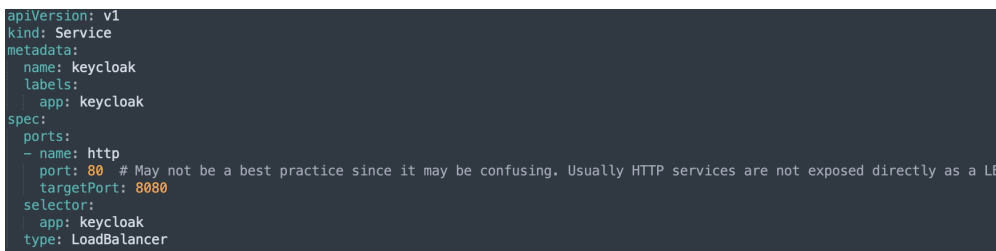


Figure 4.15: Keycloak Service YAML

By default it uses an embedded H2 database that uses a file in the filesystem, but it's not production ready and suffers many security flaws, so I configured it

to use a database within the MySQL server to store all the data it needs. This is fundamental for the persistence of the data and allows for scalability, since many instances of Keycloak may now be deployed having a single source of truth.

4.10 Egress Gateway configuration

The Egress Gateway was added to give completeness to the architecture. For this reason, a call to an external site was added from within the application.

It is configured through a *Gateway* resource and in order to define the exiting routes it also needs a *Destination Rule* and a *Virtual Service*.

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: istio-egressgateway
spec:
  selector:
    istio: egressgateway
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "*" # Using this every host can use the gateway
```

Figure 4.16: Egress Gateway YAML

In particular, the Virtual Service sets up two routes: from within the mesh to the gateway and from the gateway to the external service, defined through the host and the destination port, as represented in figure 3.10

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: worldtimeapi-through-egress-gateway
spec:
  hosts:
  - worldtimeapi.org
  gateways:
  - istio-egressgateway
  - mesh
  http:
  - match:
    - gateways:
      - mesh
      port: 80
    route:
    - destination:
        host: istio-egressgateway.istio-system.svc.cluster.local
        subset: worldtimeapi
        port:
          number: 80
        weight: 100
  - match:
    - gateways:
      - istio-egressgateway
      port: 80
    route:
    - destination:
        host: worldtimeapi.org
        port:
          number: 80
        weight: 100
```

Figure 4.17: Egress Gateway Virtual Service YAML

It should also be noted that, for better security, changing the `OutboundTrafficPolicy.mode` field in Istio's `configmap`, services inside the mesh have been forbidden to contact external services unless they are part of the service registry [6, External services].

The entries of this registry are defined precisely with the **Service Entry** resource which defines the host, the port, the type of protocol, whether the service is internal or external to the mesh and how the host must be resolved (typically using DNS).

A Service Entry was therefore created for the external service so that this outbound communication was not blocked by Istio.

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: worldtimeapi
spec:
  hosts:
  - worldtimeapi.org
  ports:
  - number: 80
    name: http-port
    protocol: HTTP
  location: MESH_EXTERNAL
  resolution: DNS
```

Figure 4.18: WorldtimeAPI Service Entry

As you can see, this policy greatly increases security because if a pod is compromised it cannot contact external services arbitrarily, thus reducing an attacker's room for maneuver.

Chapter 5

PoC results validation

After everything's set up, we need to analyze the infrastructure and its characteristics to sum up the obtained results. The main results are to be seek in whether the Zero Trust paradigm has been correctly applied or not, then we can investigate about the correctness of the other components like the Ingress Gateway, the Egress Gateway and Keycloak.

These components are treated as services within the Service Mesh, so also for them the Zero Trust is verified in the first paragraph.

5.1 Zero Trust paradigm correctness

Summarizing the Zero Trust principles we can analyze if they are fulfilled or not:

- Verify always.
- Centralized management.
- Least privilege and default deny.
- Full visibility.

In every communication between two services the proxies are the ones actively connected and they **verify** by default the authentication and authorization policies *before* putting those two services in contact. Furthermore, since they use **mutual** TLS they both exchange their identity using a certificate called SPIFFE, which is basically an x509 certificate, so that each of them knows the identity of its interlocutor. We can verify that a mTLS connection is taking place sniffing the traffic between two services using *tcpdump*, included in every Envoy proxy's container for debugging purposes, through which we can see the encrypted traffic.

For example, this is the response from the Details service to the Product Page one, as they work to respond to a normal request made from a user to the application.

This is the traffic sniffed with *tcpdump* listening from the Envoy container in the Product Page service's pod:

```
18:45:44.811813 IP details-v1-846c4d56d9-zvzsd.8082 > 172-17-0-18.productpage.default.svc.cluster.local.36176: Flags [P.], seq 1:1165, ack 1300, win 501, options [nop,nop,TS val 2295/7449e ecr 105912009e], length 1164
```

Figure 5.1: Encrypted response example

Here we can see that the communication between Details service and Product Page service is encrypted, this means that the mTLS **handshake** took place, thus the two services have exchanged their own certificates and each of them knows the identity of the other service. This happens before any real communication starts. In fact, if we disable the proxies the services don't use mTLS and the communication happens in clear:

```
19:24:21.961265 IP details-v1-846c4d56d9-zvzsd.8082 > 172-17-0-18.productpage.default.svc.cluster.local.36176:
HTTP/1.1 200 OK
date: Fri, 11 Jun 2021 19:24:21 GMT
content-length: 71
content-type: text/plain; charset=utf-8
x-envoy-upstream-service-time: 26
server: istio-envoy
x-envoy-decorator-operation: details.default.svc.cluster.local:80/*
[{
  "author": "Andrea Scoppetta",
  "publisher": "Blue Reply",
  "abstract": "Parla di un tesista in Blue Reply",
  "releaseYear": "2022"
}] [!http]
```

Figure 5.2: Clear text response example

Obviously this example can be repeated in every communication between services, changing the place from where we start sniffing with *tcpdump*. This proves that using the proxies the services are always verifying each other, so in every communication they both know each other's identity, as the **verify always** principle states. Instead, this doesn't happen when the communication happens without the proxies, so directly between the microservices, because the connection is in clear text. It's the only necessary test because for this principle we only need to prove that the services verify each other and this happens as a consequence of the handshake.

For every communications here are summarized the authentication and authorization phase:

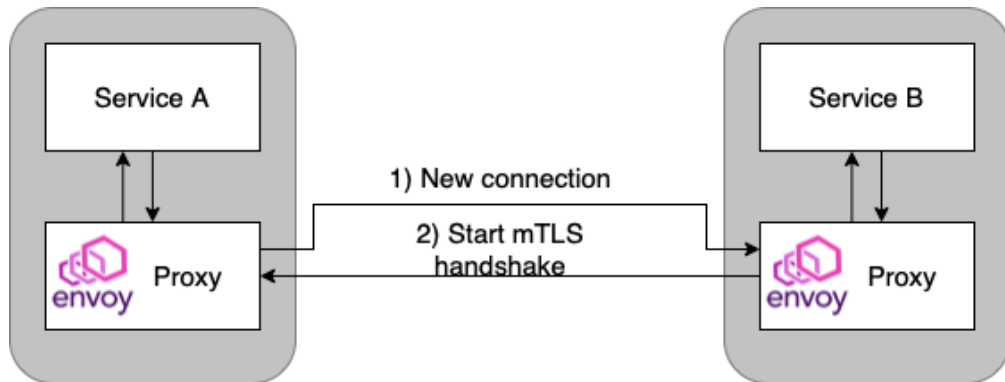


Figure 5.3: Service to Service Authentication steps

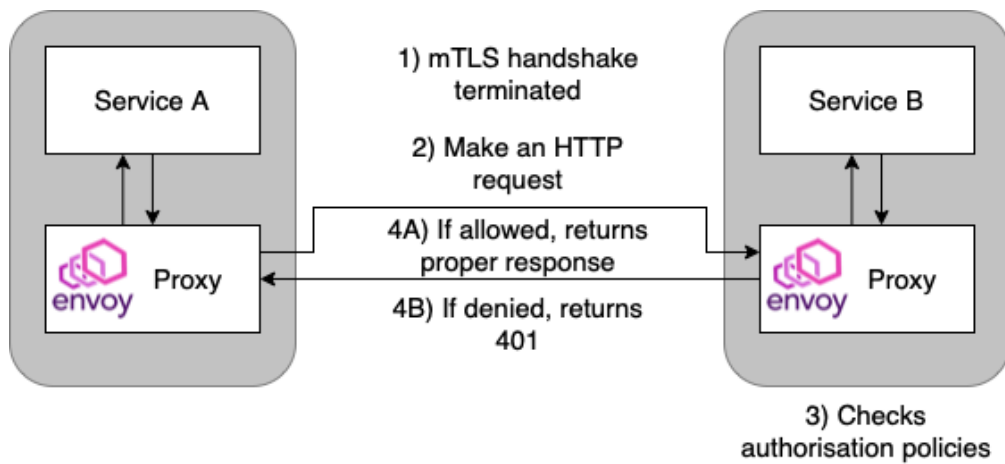


Figure 5.4: Service to Service Authorization steps

Regarding the centralizing management, remembering the picture 2.2 the control plane is the **central** component that receives all the configuration from the Service Mesh admin and then propagates them to the proxies in the data plane.

Hence, the first two principles are verified by default by Istio structure so let's move onto the other two.

Least privilege and default deny is a principle for which every component must be able to access only the information and resources that are necessary for its legitimate purpose. In our case, every microservice can only interact with the one allowed by the application structure, depicted in figure 3.7. For example if we take Reviews microservice as a reference, it *must* only talks to Ratings microservice and to the MySQL server because its logic depends only on them. Since the application does not need that Reviews speaks to Details, that route has been blocked with an Authorization Policy. If it tries to interact with Details microservice, the proxy of

the latter would deny that connection because of the configuration made by the *Authorization Policies*.

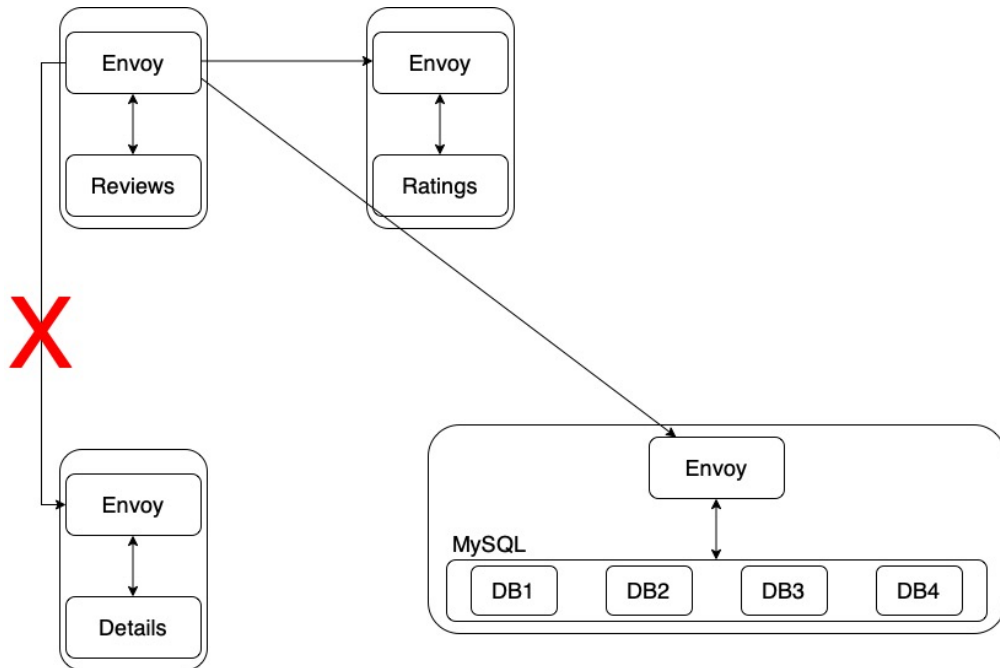


Figure 5.5: Least privilege example

Here we can see the tests made to verify the example above:

```
bash-4.4$ echo "Ratings service response = $(curl http://ratings:8083/ratings/1)"
% Total % Received % Xferd Average Speed Time Time Time Current
Dload Upload Total Spent Left Speed
100 1 0 1 0 0 2 0 --:--:-- --:--:-- --:--:-- 2
Ratings service response = 5
bash-4.4$ echo "Details service response = $(curl http://details:8082/details/Tesista)"
% Total % Received % Xferd Average Speed Time Time Time Current
Dload Upload Total Spent Left Speed
100 19 100 19 0 0 422 0 --:--:-- --:--:-- --:--:-- 422
Details service response = RBAC: access denied
```

Figure 5.6: Least privilege test

The test is executed from **Reviews** service. To test the connection from Reviews service to Ratings and Details services, I used the `curl` command to execute a GET request while the `echo` command is only used to add information to the response. As stated in figure 5.5, the request correctly receives a response from the Ratings service, in this case the stars of the review with id 1, while the Details service blocks that request. RBAC stands for Rule Based Access Control, it means that the request has been blocked by an Authorization Policy.

As another step for the test, this time I tried to make a DELETE request from Reviews to Ratings but even if the communication between the two services is allowed, this REST verb isn't, so the request is blocked.

```
bash-4.4$ echo "Reviews service response - DELETE = $(curl -X "DELETE" http://ratings:8083/ratings/1)"
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           Dload  Upload  Total      Spent    Left     Speed
100    19    100    19    0    0    413    0  --:--:--  --:--:--  --:--:--  413
Reviews service response - DELETE = RBAC: access denied
```

Figure 5.7: Least privilege test part 2

Every request that doesn't match an ALLOW policy is denied, so the caller and the REST verb must be exactly the ones already authorized. This is exactly what we expected. This test can be repeated for every microservice that must follow the application topology depicted in figure 3.7, trying for every service to contact a service to which it is connected and a service to which it isn't. Definitely, to prove this principle we need to prove that the services can only contact the services they need, using the minimum set of REST verbs they strictly need to use. This is exactly what we proved in this test.

The last principle is one of the main strength of Istio and the Service Mesh in general. **Visibility and observability** are key features and are made possible by the proxies that intercept every request and send the necessary data to the control plane so that it could elaborate metrics and provide those information via standard or custom dashboards. Here's an example of a Prometheus dashboard, a third party analytic tool that can also be customised:

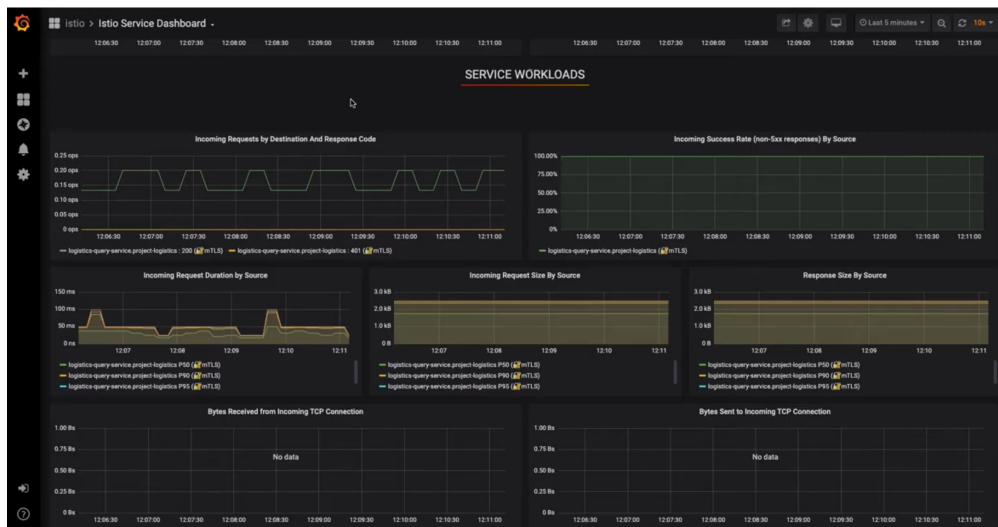


Figure 5.8: Prometheus dashboard example

Other than that, if needed, every single request can be investigated to check if there are anomalies or can be collected and sent through a pipeline that process them in a custom way to provide further information:

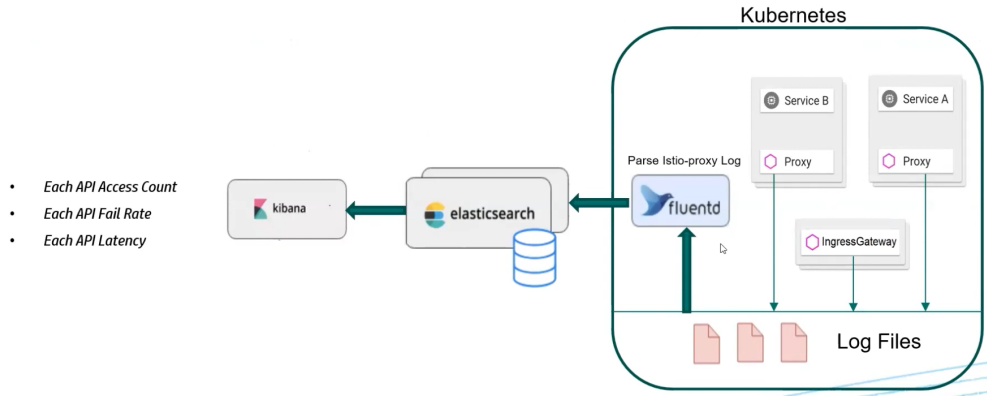


Figure 5.9: Custom observability pipeline example

Ultimately, we have verified all of the four principles and we can affirm that, using Istio, we have been able to create a Zero Trust architecture in a cluster.

5.2 Ingress Gateway correctness

The Ingress Gateway gives the external users a way to access to the application inside the cluster, providing also authentication and authorization through the use of an access token in the form of a JWT.

In this case we have to check that:

- The routing from the Gateway to the proper microservice is correct
- The authorization and authentication steps are done and are correct

The routing can be verified by checking the logs of the Ingress Gateway and, banally, checking that we get the correct response from the application after making a request to it.

The Ingress Gateway logs every incoming request keeping track of, among the other things, the HTTP method, the path, the protocol and the response code. The application logs every incoming request too. Hence, we can make a simple GET request and check if the Product Page microservices gets it and if the response code is **200**.

```

> curl -H "Host: myapp.com" \
-H "Authorization: Bearer $ADMIN_TOKEN" \
-sSl -o /dev/null -w "%{http_code}" http://myapp.com/api/v1/products/Tesista
200
2022-09-21T21:32:49.835488Z debug envoy rbac enforced allowed, matched policy ns[istio-system]-policy[allow-admin-get-book-info]-rule[0]
[2022-09-21T21:32:49.8342] "GET /api/v1/products/Tesista HTTP/1.1" 200 - via_upstream - "-" 0 781 663 661 "172.17.0.1" "curl(7.68.0)" "ef7e3a2d-5c83-958d-8ba5-086b85226a23"
"myapp.com" "172.17.0.18:8080" outbound[8080]|productpage.default.svc.cluster.local 172.17.0.7:59876 172.17.0.7:8080 172.17.0.1:8832 - -

```

Figure 5.10: Ingress Gateway logs - Authorization policy example

While, for the authorization and authentication steps we need to use a custom JWT that specifies the user's role and we also need a JWKS uri to check its signature. This can be done using Keycloak, its correctness will be discussed later.

After setting Keycloak up, we need to get an access token making a request to its exposed service. Once we have the JWT, we are going to make a request to the Ingress Gateway and check its response, in this case we are only interested in the response code.

The following cases will return **200 Success**:

Role	Method	Path
Admin	GET	/products
Admin	POST	/details
User	GET	/products
User	POST	/reviews
None	GET	/products

While every other case would return **403 Forbidden**, as we can see in this test where the [POST] /details endpoint is called using a *User* token:

```

> curl -X POST -H "Host: myapp.com" -H 'Content-Type: application/json' \
-H "Authorization: Bearer $USER_TOKEN" \
-sSl -o /dev/null -w "%{http_code}" http://myapp.com/api/v1/details -d '{"title": "Tesista2", "author": "Andrea Scoppetta",
"publisher": "Blue Reply", "abstract": "Parla di un tesista in Blue Reply", "releaseYear": 2022}'
403

```

Figure 5.11: Error 403 test

5.3 Egress Gateway correctness

The Egress Gateway plays a little role in the architecture, it is used only by the Product Page and only to make a call from it to a single external service. Thus, the test is quite simple since we only need to check if the service has been called successfully and if the Gateway has taken part in this process.

To check if the external service gets called we need to remember that Product Page make a call to it to obtain the current local time for a given timezone as a JSON and then returns it inside the response to the client. So, we only need to check if the response is complete and contains that part:

```
> curl -H "Host: myapp.com" http://myapp.com/api/v1/products/Tesista | jq
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           0         0     0    0         0     0         0    0
100  781    0  781    0    0    1597     0  --:--:--  --:--:--  --:--:-- 1597
{
  "bookDTO": {
    "title": "Tesista",
    "details": {
      "author": "Andrea Scoppetta",
      "publisher": "Blue Reply",
      "abstract": "Parla di un tesista in Blue Reply",
      "releaseYear": 2022
    }
  },
  "reviews": [
    {
      "review": "Molto bello, speriamo che finisca bene.",
      "stars": 5,
      "reviewAuthor": "Giovanni Rossi"
    },
    {
      "review": "Stupendo.",
      "stars": 5,
      "reviewAuthor": "Giovanni Verdi"
    },
    {
      "review": "Un po' bruttino.",
      "stars": 2,
      "reviewAuthor": "Andrea Verdi"
    }
  ]
},
  "serverTimestamp": {
    "abbreviation": "CEST",
    "datetime": "2022-09-21T23:53:44.290371+02:00",
    "day_of_week": 3,
    "day_of_year": 264,
    "dst": true,
    "dst_from": "2022-03-27T01:00:00+00:00",
    "dst_offset": 3600,
    "dst_until": "2022-10-30T01:00:00+00:00",
    "raw_offset": 3600,
    "timezone": "Europe/Paris",
    "unix_time": 0,
    "utc_datetime": "2022-09-21T21:53:44.290371+00:00",
    "utc_offset": "+02:00",
    "week_number": 38
  }
}
```

Figure 5.12: External service test

Instead, to check if the Gateway has forwarded the request to the external service we can watch its logs, like we did for the Ingress Gateway. They contains every request that goes from the Egress Gateway to the outside, with information about the destination:

```
[2022-09-21T22:02:01.272Z] "GET /api/ip HTTP/2" 200 - via upstream - "-" 0 394 426 425 "172.17.0.18" "Java/11.0.14.1" "4ef0e254-552d-9022-93e2-a14ce8343d82" "worldtimeapi.org" "213.188.196.246:80" outbound|80|worldtimeapi.org 172.17.0.19:60346 172.17.0.19:8080 172.17.0.18:56622 - -
2022-09-21T22:02:03.261353Z debug envoy http async http request response headers (end_stream=true):
  ':status', '202'
  'vary', 'Origin'
  'date', 'Wed, 21 Sep 2022 22:02:03 GMT'
  'content-length', '0'
  'x-envoy-upstream-service-time', '0'
```

Figure 5.13: Egress Gateway - External service logs

5.4 Keycloak correctness

Keycloak, despite all of its features, is used as an authentication server so it only keeps track of the users and provides them an access token after a successful authentication.

We need to add three main things in order to provide a JWT:

- *A client*: the application that needs an authentication server.
- *The roles*: the roles that such client can handle.
- *The users*: the registered users for that application, which can assigned to a certain role and are composed by username and password.

Everyone of these aspect can be configured using Keycloak's dashboard so we have an immediate feedback if we've done them right.

Regarding the JWT, it can be obtained through a request containing all the user's credentials and it can be verified on the website <https://jwt.io> where the JWT get decoded and is shown as a JSON. We can then check if the custom fields are correct, in this case the roles of that particular user. For example here's a snippet of a decoded JWT:

```
"resource_access": {
  "bookinfo": {
    "roles": [
      "ADMIN"
    ]
  },
  "account": {
    "roles": [
      "manage-account",
      "manage-account-links",
      "view-profile"
    ]
  }
},
"scope": "openid email profile",
"sid": "f4725dcd-e34d-4632-b047-8948d80b1ee7",
"email_verified": false,
"name": "Andrea Scoppetta",
"preferred_username": "andreasco",
"given_name": "Andrea",
"family_name": "Scoppetta",
"email": "andre.sco@icloud.com"
}
```

Figure 5.14: Decoded JWT example

Where the noticeable fields are the `resource_access.bookinfo.roles` and the user's details like name, username and email.

Furthermore, we can check if using that JWT we can access the application as expected. In this way we would check the correctness of the signature of the JWT and that the JWKS uri works fine since the Ingress Gateway validate the signature every time a user makes a request.

Chapter 6

Real case scenario: Upgrading to a Service Mesh

During my internship in Blue Reply I was able to really dive into the topic of the Service Mesh and in particular how it can be used to obtain a Zero Trust Architecture. Until now we've talked about a comprehensive PoC, made in order to underline all of the Service Mesh features but the final part of the internship allowed me to get my hands on a real application, called *TUAMH*, used in a critical workflow of a big company.

Briefly, the core of the application was deployed as a set of microservices in a cluster and the Zero Trust Architecture was implemented using the initial request's JWT for every service to service connection. This led to more utility code, less homogeneity between the microservices and more work from the side of the developers, and nevertheless this solution was still less secure than the one with a Service Mesh because there was no encryption and the mutual authentication had to be implemented manually.

In this chapter will be explained the initial general architecture of the application, its flaws and a solution to them, and lastly how the initial architecture was *upgraded* thanks to the Service Mesh.

6.1 Components description

Before starting to describe the whole architecture, here is a brief introduction of its main components.

- *Azure Application Gateway Front-end*: is the gateway that exposes the User Account's services on internet.
- *Azure Kubernetes Services (AKS)*: is a fully managed container orchestration

service that can be used to deploy, scale and manage Docker containers applications in a cluster environment.

- *Back-end for Front-end (BFF) Microservices*: is a cluster of microservices that works as a middleware between the client and the identity access on-premises infrastructure.
- *jBPM*: is an open-source workflow engine that can automate business processes and decisions.
- *Traefik*: is the cluster's Ingress Controller, in charge of managing the incoming HTTP requests and routing them to the services. In conjunction with OPA, Traefik acts as API gateway, and more specifically as **policy enforcement point**, blocking unauthenticated or unauthorized requests.
- *OPA*: Open Policy Agent, open-source component to manage authentication and authorization policies. It acts as **policy decision point** in conjunction with Traefik, checking the security tokens from the incoming requests and evaluating authentication and authorization for the protected APIs.
- *Core Microservices*: is a cluster of Microservices the implements the User Account business logic.

6.2 Architecture description

In figure 6.1 and 6.2 the User Account Architecture is described from respectively logical and technical point of view. At high level, User Account can be divided into two big logic blocks: **Microsoft Azure** and **IAM on-premises**. These two main infrastructure components are in completely separated network segments and communicate over https. Let's explain the role of each component of both Azure and IAM on-premises blocks.

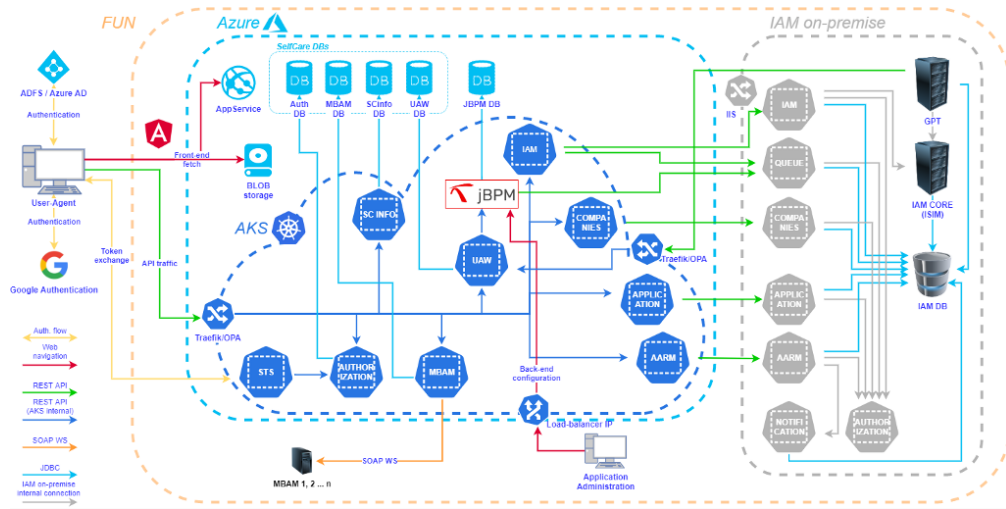


Figure 6.1: User Account Logic Architecture

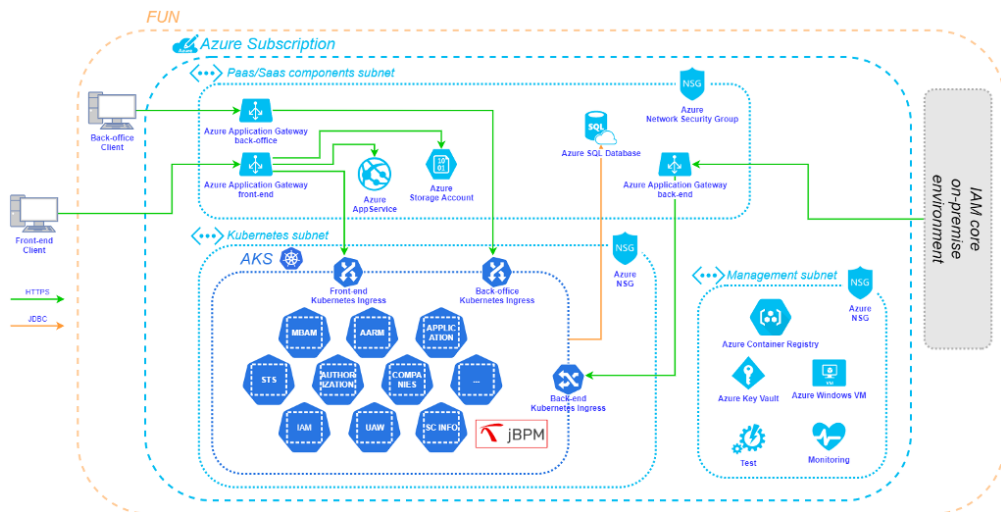


Figure 6.2: User Account Technical Architecture

6.3 Azure

Starting from the upper left corner of the figure 6.2, the first component is the **Azure Application Gateway**. It is the web traffic load balancer that manages the traffic from and to the User Account Application. Depending on whether the request path begins with /api or not, the Azure Gateway routes it to the **BFF** microservices or to the **Azure AppService** respectively.

Azure AppService is the User Account's web hosting service that serves the (Angular-based) User Account single-page application. All requests to BFF microservices pass through the **Traefik-OPA** gateway which, before routing them, performs the authentication and authorization checks. More precisely: it first verifies that the request has a valid (Public or User) authorization token and then that the client has sufficient grants to access the service (i.e. the token must include the scope valid for the requested API).

The main role of the **BFF microservices** is to operate like a middleware between the client and the REST IAM Core (with some exceptions where the BFFs return directly without routing to the Cores). When a request comes, they first verify its validity (by validating the access token's signature), and then, if needed, they process it (also by the support of a fully dedicated database, as you can see in the figure 6.1), optionally route it to the IAM Core and, finally, produce and return a response to the client.

Another component deployed in the AKS environment is **jBPM**. It is an open-source, flexible Business Process Management (BPM) suite that offers a complete authoring, execution, management and monitoring environment to support the full life-cycle of workflow processes. The Approval workflow is triggered by the submission of a user request (e.g. Account creation or Account suspension). Based on the data received, the process is able to determine the next step in which the request must be placed (Validation, Approval or Declaration). Every request to jBPM must come from *UAW BFF*, which is a particular microservice in charge of exposing the UAW capabilities and brokering the security context between TUAMH and jBPM. Outgoing requests from jBPM to external services (i.e. IAM Core services) must be routed through UAW BFF as well. jBPM also offers a management dashboard to edit, configure, deploy and administer the bpm workflows. The cluster exposes such interfaces through a separate interface (back-office) with no AA enforcement: in order to access the dashboard, users must authenticate directly on jBPM.

6.4 IAM on-premise

In the previous subsection, we have already mentioned the **Core Microservices**. As the name suggests, they implement the **IAM Core Service**, a full, cross application, web API based interface exposing company's IAM services and capabilities. In order to do this, they are heavily integrated with **IBM Security Identity Manager (ISIM)**, which is the core product of the company's IAM system.

Many TUAMH's features rely on IAM Core Service, but it is important to clarify that TUAMH is a client external to IAM, which must be authenticated as any other third-party client.

Finally, figure 6.1 includes the IAM dashboard **Global Provisioning Tool (GPT)** because some UAW administrative operations are available on it. Therefore, GPT acts as a technical client of TUAMH services, authenticating accordingly.

6.5 Architectural flaws

All the custom microservices that compose the described architecture are written in **Spring Boot** using the **Spring Security** module. The user obtains a JWT after the authentication and it has to be used for every subsequent request, this token is then forwarded from microservice to microservice for every internal request in order to try to create a **Zero Trust architecture**.

As we said about Zero Trust principles, this implementation fulfills only one of them: *verify always*. With this implementation is difficult to implement more principles without a complete change in the architecture with a consequent effort from the infrastructure and development team.

Unfortunately this was the main way to implement a Zero Trust architecture before the Service Mesh took hold, so an "upgrade" was needed.

Thanks to the use of it, as we have largely seen before, the Zero Trust architecture can be fully implemented, even in an existing architecture without a complete refactoring. The main steps were:

- Install Istio and activate the Egress Gateway.
- Disable JWT forward and check in every microservice.
- Configure Istio and the Egress Gateway.

6.6 Upgraded architecture

After Istio installation the architecture has slightly changed. Every microservice is now coupled with an **Envoy** proxy allowing the **centralized management** and **full visibility** principles, the internal REST API uses **mTLS** and every communication the on-premise part is done through the Egress Gateway. Even in this case, though, the connection to the databases need to be remain direct because of the already discussed problems.

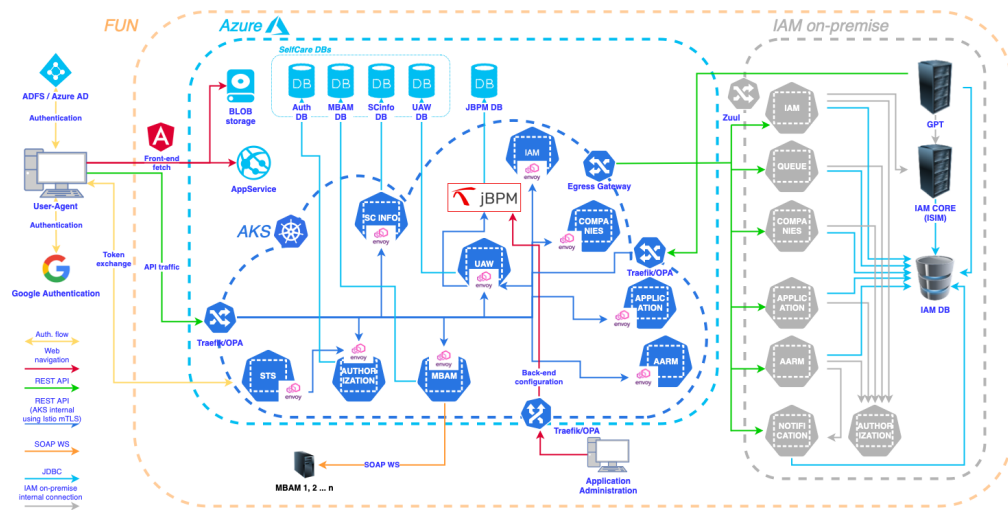


Figure 6.3: Logic Architectures using Istio

Now, we need to disable Spring Security in every microservice. That is done by simply removing the annotation from the main class and commenting the JWT checks in every microservice controller. After this, the service to service communications happen without any JWT exchange and instead are made by the proxies so they are encrypted and **mutually authenticated** using certificates, following the **verify always** principle.

Then, in order to configure Istio's security we can proceed in a similar way as we did before during the implementation of the PoC. The Ingress Controller don't have to be modified, the only changes happen inside the mesh. First of all, every routing rule implemented in Traefik need to be converted in an Istio one, using *Virtual Services* in order to properly sort the request to the appropriate BFF service. Then, every service to service communication has to be enforced with an *Authorization Policy*, we can define which service can interact with a certain service and how the request can be done, following the **least privilege and default deny** principle.

The last configuration step would be to grant access to the on-premise part of the architecture. This part could not be modified because it resides on the company's server to must be treated as an external service thus can be contacted through the **Egress Gateway**. As in the PoC, every other outgoing connection has been blocked and the only allowed ones are the ones to the company servers. This is done again through the *Service Entries*.

Chapter 7

Real case results validation

The tests to check whether the Zero Trust paradigm is satisfied or not are quite similar to the ones done in the case of the PoC in the chapter 5. In this case we need to check the correctness of the Zero Trust paradigm, the Ingress Gateway and the Egress Gateway

7.1 Zero Trust paradigm correctness

As we've said multiple times, the correctness of the paradigm is based on verifying the four pillars that it is made of: verify always, centralized management, least privilege and default deny and full visibility. Like we said for the PoC, the only pillars we have to verify are **verify always** and **least privilege** because the other ones are already satisfied by design because of the behaviour of Istio. To check if the proxies really take part in the transaction we can check if mTLS is being used, that is if the connection is encrypted. Using *tcpdump* and listening to the connection between the **STS** and the **Authorization** services, two of the most important ones, we can see that without injecting the proxies the connection is in clear text:

The image shows a network capture snippet. At the top, two callouts identify the 'STS service' and the 'Authorization service'. The main text is an HTTP response from the STS service to the Authorization service. The response is in plain text and contains a JSON object with user identity information.

```

14:09:51.360085 IP sts-v3-106c8a52l2-qvfua.10012 > 168-10-0-28.authorization.iam.svc.cluster.local.91276:
HTTP/1.1 200 OK
date: Fri, 11 Jun 2021 14:09:51 GMT
content-length: 240
content-type: text/plain; charset=utf-8
x-envoy-upstream-service-time: 26
server: istio-envoy
x-envoy-decorator-operation: sts.default.svc.cluster.local:80/*
[
  {
    "uid": "FCCWZZ",
    "name": "MARIO",
    "surname": "ROSSI",
    "employeeType": "C",
    "sectorCode": "01",
    "companyCode": "101312",
    "servizioFeed": "MAN",
    "idGoogle": "mario.rossi@adm. ....com",
    "cdc": "0008679500",
    "identityType": "INTRANET_IDENTITY"
  }
]

```

Figure 7.1: Clear text response example

Instead, injecting the proxies the connection is immediately, and by default, upgraded to an mTLS connection where the both parties know the identity of each other:

The image shows a network capture snippet. At the top, two callouts identify the 'STS service' and the 'Authorization service'. The main text shows the start of an encrypted connection between the two services.

```

12:29:14.290823 IP sts-v3-106c8a52l2-qvfua.10012 > 168-10-0-28.authorization.iam.svc.cluster.local.91276: Flags [P.], seq 1:1165, ack 1300, win 501,
options [nop,nop,TS val 2295774496 ecr 1059120096], length 1164

```

Figure 7.2: Encrypted response example

This guarantees that the mTLS *handshake* has taken place, so the services have *verified* each other before instantiating the connection.

While, for the least privilege property we can check if the STS service can communicate with the Authentication service but not with the Application one. Here's the test results making a `curl` request from the STS pod to the other two services:

```

bash-4.4$ echo "Authorization service response =\n$(curl http://authorization:10012/api/openid/token | jq)"
Authorization service response =
[{"
  "jti": "22494a49-8411-420f-95df-82f2e62f3ab3",
  "ses": "e279aae8-1b19-48a9-9d34-d79d1cd5dc1e",
  "sub": "e279aae8-1b19-48a9-9d34-d79d1cd5dc1e",
  "usn": "e279aae8-1b19-48a9-9d34-d79d1cd5dc1e",
  "iss": "https://test-useraccount. [REDACTED]. com/sts-on-cloud/oauth2/v1",
  "aud": "https://test-useraccount. [REDACTED]. com/",
  "aut": "https://test-useraccount. [REDACTED]. com/sts-on-cloud/oauth2/v1",
  "iat": 1603375748,
  "nbf": 1603375688,
  "exp": 1603379348,
  "scp": [
    "PUBLIC"
  ]
}]
bash-4.4$ echo "Application service response = $(curl http://application:8080)"
Application service response = RBAC: access denied

```

Figure 7.3: Least privilege test

Remember that this is possible thanks to the use of the *Authorization Policies* that are configured for this specific architecture. Through them, the proxy on the receiving end of the communication decides which services can allow.

Finally, also in this case we can confirm that using Istio the Zero Trust paradigm is fulfilled.

7.2 Ingress Gateway correctness

Again, the tests we can make on the Ingress Gateway are to check if we can correctly reach the services inside the mesh and if the Gateway logs the requests we are making.

Furthermore, we need to check if the JWT validation and authorization policies are correct, checking if using the right JWT we can reach the proper service.

Thus, we can make a valid request to the main application using a correct JWT to check if the request is properly routed and if the JWT is really validated as we thought:

```

curl -H "Authorization: Bearer $TOKEN" \
  -sSl -0 /dev/null -W "%{http code}" https://useraccount. [REDACTED]. com
200%

```

Figure 7.4: Curl success test

Inspecting the Ingress Gateway logs we can see that the request is received, matches an Authorization Policy that allows it and then it's properly forwarded to the Application service:

```
2022-09-30T17:22:19.432387Z debug envoy rbac enforced allowed, matched policy ns:istio-system/policy/allow-user-get-user-account-rules10
[2022-09-30T17:22:19.432Z] "GET / HTTP/1.1" 200 - via upstream - "0 781 663 661" 172.17.0.1 "curl/7.68.0" "bf73a0-9b13-91ae-712a-271938101318za2"
"useraccount. ██████████.com" "172.17.0.18:91276" outbound|91276|authorization.iam.svc.cluster.local 172.17.0.7:59876172.17.0.7:91276 172.17.0.1:8837 - -
```

Figure 7.5: Ingress Gateway logs

As a final test we can check that using a wrong JWT the result is **403 Forbidden**:

```
curl -H "Authorization: Bearer $WRONG_TOKEN" \
-sSI -0 /dev/null -W "%{http_code}" https://useraccount. ██████████.com
403%
```

Figure 7.6: Error 403 test

7.3 Egress Gateway correctness

The Egress Gateway plays a role a bit different from the PoC because in this case it calls a service external to the mesh that's not on the internet but inside the on-premise part of the architecture, that resides on the company's servers. The services on the on-premise part are mostly **core** microservices, for example the Application microservice on the Azure cloud receives the requests and then contact the Application microservice on the on-premise part to execute its core logic.

What we can do to test its functionality is to make a `curl` request from the Envoy proxy in the Application's pod to the Application service on the on-premise part and check if the response is correct and if the Egress Gateway routed the request, inspecting its logs.

```
bash-4.4$ echo "Authorization core service response =\n$(curl http://internal.authorization:9012/api/v1/user/account | jq)"
Authorization core service response =
[{"uid": "FCCCWZZ",
"name": "MARIO",
"surname": "ROSSI",
"employeeType": "C",
"sectorCode": "01",
"companyCode": "101312",
"servizioFeed": "MAN",
"idGoogle": "mario.rossi@adm. ██████████.com",
"cdc": "0008679500",
"identityType": "INTRANET_IDENTITY"}]
```

Figure 7.7: Curl on-premise service

The Egress Gateway logs confirms that the request has passed through it:


```
2022-09-30T18:15:31.7363117 "GET /api/v1/user/account HTTP/2" 200 - via upstream - "-" 0 247 172 419 "172.17.0.18" "Java/11.0.14.1"
"1e70e514-192a-5182-1731-17ae912f1a11" "internal.authorization" "215.118.151.223:80" [outbound] [internal.authorization] 172.17.0.19:60346
172.17.0.19:8080 172.17.0.18:56622 - -
2022-09-30T18:16:03.7363117 debug envoy http async http request response headers (end_stream=true) :
'status', 202'
'vary', 'Origin'
'date', 'Fri, 30 Sep 2022 18:15:31 GMT'
'content-length', '0'
'x-envoy-upstream-service-time', '0'
```

Figure 7.8: Egress Gateway logs

Chapter 8

Conclusions and future works

The objective of this thesis was to analyze the implementation of a **Zero Trust architecture** in a **Kubernetes environment**, the de facto container orchestration standard to which an increasing number of online applications are migrating, in order to understand how to design it to obtain and manage *workload authentication*, in particular using a Service Mesh.

This was done in three main steps: first of all I studied all the documentation about **Zero Trust security**, **Service Mesh** and in particular **Istio**, then I moved onto the design of a **Proof of Concept** to prove what Istio is capable of and to confirm that a Zero Trust architecture can be achieved thanks to it. This phase included also the appropriate tests to prove my initial goal. In the end, when I had the certainty about the level of security and the benefits that the infrastructure had, I moved onto a **real case scenario**.

It consisted of an already up and running application, serving a critical workflow in a big company, customer of Blue Reply, my internship company. This application had already implemented the Zero Trust paradigm but in an "older" way, without the use of the Service Mesh. This led to some flaws, already analyzed, that have been overcome thanks to Istio. I then designed the changes in the architecture and implemented Istio, deactivating the older mechanisms for the Zero Trust. As expected, the tests showed big improvements both in security and observability, with the satisfaction of the customer.

In conclusion, I've given two proofs that the Service Mesh **can** help to achieve a Zero Trust architecture in a fairly easy way, adding also extra features to the cluster like observability and traffic management capabilities. Especially the real case scenario proves that already existing applications can be upgraded to gain higher levels of security with less effort than before.

This, however, is only a first step in the ever-evolving world of cloud computing, for example the **multicloud** approach is rapidly gaining ground. Briefly, it consists in spanning an application between various cloud service providers and we can already foresee all the challenges that this may create.

An extension of this thesis may be, for example, a deeper analysis of this incoming approach, to understand if a Zero Trust architecture can be implemented in that situation and if so, how to do that. Istio is designed to work across multiple cloud providers so expand my work to the multicloud approach could be very interesting.

Furthermore, Istio lately is introducing **Ambient Mesh**, a new data plane mode that's designed to simplify operations, widen application compatibility, and reduce infrastructure cost. With it, users can keep Istio's essential functions like Zero Trust security, telemetry, and traffic management while forgoing sidecar proxies in favor of a mesh data plane that is embedded into their infrastructure. Briefly, the new data plane is not made by a proxy for each service but a proxy for *a group* of services, leading to computational resources reduction and performance improvements. This may be the next step of the Service Mesh, as many insiders say, so may be worth to start learning this new paradigm and understand which real benefits it can bring.

Bibliography

- [1] *Virtual Service*. <https://istio.io/latest/docs/reference/config/networking/virtual-service>. Accessed: 21-09-2022 (cit. on p. 14).
- [2] *Kubernetes Service Mesh: A Comparison of Istio, Linkerd, and Consul*. <https://platform9.com/blog/kubernetes-service-mesh-a-comparison-of-istio-linkerd-and-consul>. Accessed: 20-09-2022 (cit. on p. 23).
- [3] *Run Kubernetes Using Minikube Cluster on The AWS Cloud*. <https://aws.plainenglish.io/running-kubernetes-using-minikube-cluster-on-the-aws-cloud-4259df916a07>. Accessed: 18-03-2022 (cit. on p. 37).
- [4] *Step by Step Guide to install Istio Service Mesh in Kubernetes*. https://dev.to/techworld_with_nana/step-by-step-guide-to-install-istio-service-mesh-in-kubernetes-d6d. Accessed: 30-03-2022 (cit. on p. 37).
- [5] *Istio Application Requirements - Server First Protocols*. <https://istio.io/latest/docs/ops/deployment/requirements>. Accessed: 10-04-2022 (cit. on p. 38).
- [6] *Istio Egress*. <https://www.youtube.com/watch?v=MHKc4hfszUI>. Accessed: 20-04-2022 (cit. on p. 52).