# Distributed Authentication Mesh*

## A Concept for Declarative Ad Hoc Conversion of Credentials

Christoph Bühler

Spring Semester 2021
University of Applied Science of Eastern Switzerland (OST)

As more and more applications run in containerized cloud environments, securing their architectures against attackers is an important concern. Applications defend themselves against intrusion with various authentication mechanisms such as OpenID Connect. However, legacy applications that are not updated nor rewritten tend not to support modern security standards. Enabling applications to communicate with legacy (or third-party) software often requires to introduce code changes to the modern apps.

To eliminate leaking credentials (such as access tokens) and to reduce the risk of bugs, this project targets the dynamic conversion of a user identity. This identity is used to authenticate the user instead of the original credentials. This project provides the conceptional idea and the architecture, as well as a platform specific example of such a solution. A Proof of Concept answers relevant questions for the realization of such a framework. The evaluation then shows that the proposed solution is as secure as the current state of the art and validates the architecture against the goals. The conclusion provides information about the project, possible use cases, and the goals of follow-up projects.

* I would like to express my appreciation to Dr. Olaf Zimmermann for guiding and reviewing this work. Furthermore, special thanks to Florian Forster, who provided the initial inspiration and challenged the results of this project from a practical perspective.

# Contents

1 Introduction 5

2 Definitions and Clarification of the Scope 6
2.1 Scope of the Project 6
2.2 Kubernetes as an Orchestration Engine 7
2.2.1 Introduction 7
2.2.2 Terminology 9
2.3 The Operator Pattern 10
2.4 The Sidecar Pattern 12
2.5 Controlling the Data with a Service Mesh 13
2.6 Authentication, Authorization, and Security 14
2.6.1 Basic Authentication (RFC7617) 14
2.6.2 OpenID Connect (OIDC) 14
2.6.3 Zero Trust Environment 16

3 State of the Art and the Practice 17
3.1 Accessing Legacy Systems from Cloud-Native Applications 17
3.2 External Authentication and Identity Transport 19
3.3 Missing Dynamic Credential Transformation 20

4 Distributed Authentication Mesh 23
4.1 Definition 23
4.2 Goals and Non-Goals of the Project 23
4.3 Differentiation from Security Assertion Markup Language 25
4.4 Architecture of the Distributed Authentication Mesh 25
4.4.1 Federated Identity with Diverging Authentication Schemes 26
4.4.2 Conceptional Architecture 26
4.4.3 Platform-Specific Example in Kubernetes 28
4.5 Securing the Communication between Applications 39
4.6 Implementation Proof of Concept (PoC) 39
4.6.1 Case Study for the PoC 41
4.6.2 Automation Engine for Applications 43
4.6.3 Network and Routing Proxy for Communication 44
4.6.4 Translator 45

5 Evaluation 48
5.1 Architecture against Requirements 48
5.1.1 NFR 1: Improve Security 48
5.1.2 NFR 2: Secure implementation 48
5.1.3 NFR 3: Generic Usage 49
5.1.4 NFR 4: Performance Impact 50
5.1.5 NFR 5: Modularity 50

5.1.6 NFR 6: Integration into Infrastructure . . . . . . . . . . . . . . . . . 50
5.1.7 NFR 7: Scalability . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
5.1.8 NFR 8: Separation of Concerns . . . . . . . . . . . . . . . . . . . . . 51
5.1.9 NFR 9: No Data-Transfer . . . . . . . . . . . . . . . . . . . . . . . . 51
5.1.10 NFR 10: Error Handling . . . . . . . . . . . . . . . . . . . . . . . . . 51
5.2 Leaking Credentials and Developer Experience . . . . . . . . . . . . . . 52

**6 Conclusions and Outlook 54**

**Bibliography 56**

**Appendix A: Common Kubernetes Terminology 58**

**Appendix B: Installation of the PoC 60**

**Appendix C: Teaching Material for Kubernetes Operators 61**
Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
Learning Objectives . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
Kubernetes Operators and their Use . . . . . . . . . . . . . . . . . . . . . 61
What is an Operator? . . . . . . . . . . . . . . . . . . . . . . . . . . 61
How do Operators work? . . . . . . . . . . . . . . . . . . . . . . . . . 63
What is an Operator SDK? . . . . . . . . . . . . . . . . . . . . . . . . 65
Exercise: Create a Custom Operator with an SDK . . . . . . . . . . . . . . 66
TL;DR . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
Create and Run an empty Operator . . . . . . . . . . . . . . . . . . . . 66
Create the Custom Resource Definition . . . . . . . . . . . . . . . . . . 68
Reconcile the Custom Resource . . . . . . . . . . . . . . . . . . . . . . 69

## List of Tables

1 Key Kubernetes Terminology . . . . . . . . . . . . . . . . . . . . . . . . 9
2 Functional Requirements . . . . . . . . . . . . . . . . . . . . . . . . . 23
3 Non-Functional Requirements . . . . . . . . . . . . . . . . . . . . . . . 24
A.1 Common Kubernetes Terminology . . . . . . . . . . . . . . . . . . . . . 58

## List of Figures

1 Illustration of the problem with diverging authentication mechanisms . . . 6
2 Kubernetes Container Evolution . . . . . . . . . . . . . . . . . . . . . . 8
3 UML of Kubernetes Resources (partially) . . . . . . . . . . . . . . . . . 10
4 Kubernetes Operator Workflow . . . . . . . . . . . . . . . . . . . . . . . 11
5 Example of a Sidecar Container . . . . . . . . . . . . . . . . . . . . . . 12

6 OpenID Connect Code Flow . . . 15
7 Microservice Architecture with Legacy Components . . . 18
8 Current Process of Legacy Communication . . . 19
9 Service Landscape with various authentication mechanism . . . 21
10 Abstract Solution Architecture . . . 27
11 Automation with an Operator in a Kubernetes Environment . . . 29
12 Determination of the Relevance of a Deployment or a Service . . . 30
13 Automated Enhancement of a Deployment and a Service . . . 31
14 The Relation of the Public Key Infrastructure and the System . . . 32
15 Provide Key Material to the Translator . . . 33
16 Checking the Signature of the transmitted Identity . . . 34
17 Networking with an Proxy . . . 35
18 Outbound Networking Sequence . . . 35
19 Inbound Accepted Networking Sequence . . . 36
20 Inbound Rejected Networking Sequence . . . 37
21 Translator Process . . . 38
22 Component Diagram of the Case Study . . . 41
23 Sequence Diagram of the Communication in the Case Study . . . 42
24 Activity Model for Kubernetes Resources in the Automation Engine . . . 43
25 Communication with an Invalid Access Token . . . 46
26 Communication with a Valid Access Token . . . 47
C.1 Kubernetes Operator Reconciliation Loop . . . 62
C.2 Kubernetes Operator Pattern . . . 62
C.3 Parts of an Operator and their Interaction with Kubernetes . . . 63
C.4 Kubernetes Operator Workflow . . . 65

# 1 Introduction

Cloud environments like Kubernetes solve many problems such as the discovery of services and data transfer or communication between services and applications in general. As the development of cloud-native applications (CNA) evolves, older applications move to the cloud as well.

However, a specific problem is not yet solved: “dynamic” trusted communication between services. For example, a service with the capability of handling OpenID Connect (OIDC) credentials wants to communicate with a service that only knows Basic Authentication. The OIDC-capable service must implement some conversion mechanism or know static Basic Auth credentials to communicate with the basic auth service. In general, this introduces changes to the software. In small applications that consist of one or two services, implementing this conversion may be a feasible option. But if we look at an application that spans over a big landscape and a multitude of services, implementing every used authentication mechanism will be error-prone work and does not scale well[1]. In practice, we encountered the given scenario at various points in time when older applications were migrated into a cloud environment and newer applications were built around it. In almost all cases, the modern software was changed to communicate with the legacy systems, instead of modifying the legacy application.

The goal of the project “Distributed Authentication Mesh” is a concept and an architecture for a solution to the problem of dynamic credential conversion. By introducing multiple elements, such as a translator in conjunction with a proxy that is capable of modifying HTTP headers in-flight, the described problem can be solved. The proposed concept makes use of a common domain language to transfer the authenticated identity of a user between services of an application. The proxy intercepts the requests and instructs the translator to transform the common language into the valid authentication format of the destination.

The remainder of the report describes used technologies, terminology, and concepts. Furthermore, the state of the art gives an overview of the current situation and the present solutions in practice. With the description of the distributed authentication mesh, the report shows the conceptual idea and the architecture as well as a platform-specific example in Kubernetes. The feasibility of the concept is tested with the implementation of a Proof of Concept (PoC) on Kubernetes. The evaluation, following the description of the solution, validates if the goals and non-goals of the solution are met. The conclusion gives an overview of the work and a summary of the project.

For the understanding of the report, basic knowledge about Docker, Kubernetes, microservices and security is required. The implementation of the PoC is based on Kubernetes to display the concepts of the solution practically. In terms of authentication and authorization, the PoC uses OpenID Connect and Basic Authentication, which are both described in later sections.

[1] According to the matrix problem: $X$ services $* Y$ authentication methods

# 2 Definitions and Clarification of the Scope

This section provides general information about the project, the context, and prerequisite knowledge. It gives an overview of the context as well as terminology and general definitions.

## 2.1 Scope of the Project

This project addresses the specific problem of declarative conversion of user credentials, for example an access token, to ensure authorized communication between services. When multiple services with different authentication mechanisms communicate with each other, the services need to translate the credentials and send them to their counterpart. The goal of this project is to prevent user credentials from being transmitted to other services and to remove the need for code changes to transform credentials to another format.

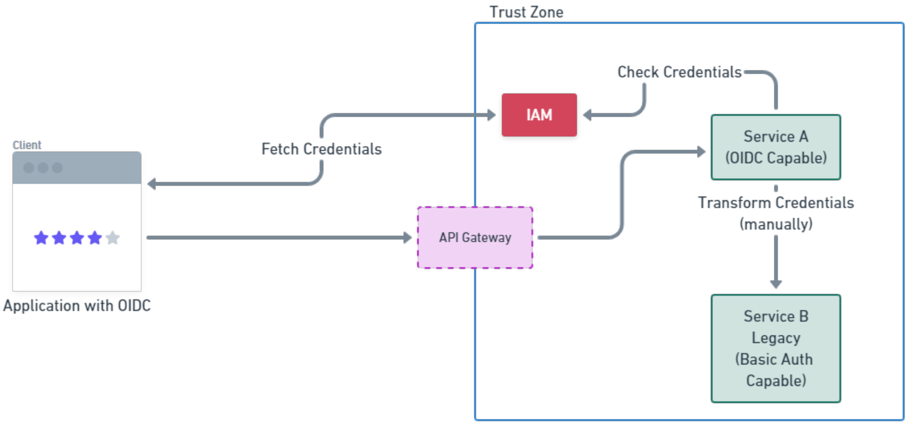

Figure 1: Illustration of the problem with diverging authentication mechanisms

Figure 1 shows an example where an automatic and dynamic translation of access credentials would be useful. Service A needs to translate the received OIDC access token to some information encoded in Basic Authentication to access Service B.

To solve the problem, an automation component enhances services that are part of the application with additional functionality. A proxy in front of the service captures in-, and outgoing traffic to modify the `Authorization` HTTP header. Additionally, a translator transforms the original authentication data into a form of identity and encodes it with a common language format. The receiving service can validate this encoded identity and transforms the identity into valid user credentials again. This automatic transformation of credentials (e.g. from OIDC to Basic Auth) replaces manual work which may introduce code changes to either service. The deliverables of this and further projects may aid applications or APIs to communicate with each other despite different authentication mechanisms.

The solution may be feasible for various platforms but to provide a practical demo application, the Proof of Concept (PoC) runs on Kubernetes. Kubernetes[2] is an orchestration platform that works with containerized applications. The PoC resides in an initial version in an open-source GitHub repository. The PoC demonstrates that it is possible to instruct an Envoy[3] proxy to communicate with an injected service to modify authentication credentials in-flight. To separate the proposed solution from more complex concepts like a service mesh, the PoC can run without a service mesh on a Kubernetes cluster and uses the built-in service discovery of Kubernetes to communicate.

## 2.2 Kubernetes as an Orchestration Engine

This section provides a general overview of Kubernetes. Kubernetes is a prominent orchestration engine that manages workloads on worker-nodes. In this project, Kubernetes is used as platform for the specific implementation example in the PoC. The solution does not require Kubernetes or any other cloud environment platform but certain features, like automation with operators, support the solution. It is possible to use other environments, such as Docker Swarm or a native implementation on an operating system, to run the proposed solution.

### 2.2.1 Introduction

Kubernetes is an open-source platform that manages containerized workloads and applications. Workloads may be accessed via “Services” that use a DNS naming system. Kubernetes uses declarative definitions to compare the actual state of the system with the expected state [1].

[2] https://kubernetes.io/

[3] https://www.envoyproxy.io/

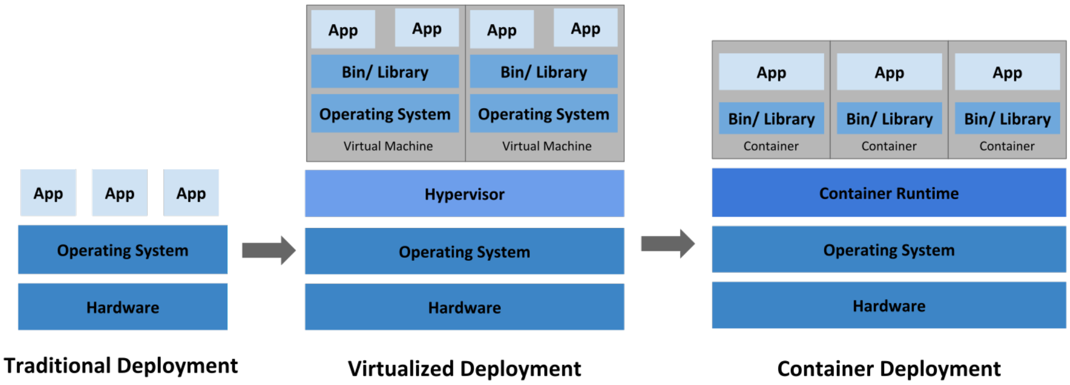

Figure 2: Container and Deployment Evolution. Description of the evolution of deployments as found on the documentation website of Kubernetes [1]. This image is licensed under the CC BY 4.0 license [2].

According to Kubernetes, the way of deploying applications has evolved. As shown in Figure 2, the “Traditional Era” was the time when applications were deployed via FTP access and started manually (e.g. on an Apache web server). Then the revolution to virtual machines came and technologies that could virtualize a whole operating system, such as VMWare, were born. The latest stage, “Container Era,” defines a new way deploying workloads by virtualizing processes instead of operating systems and therefore better use the given resources [1].

Kubernetes is a major player among others like “OpenShift” or “Cloud Foundry” in “Container Deployment” as seen in Figure 2 and supports teams with the following features according to the documentation [1]:

- **Service discovery and load balancing**: Use DNS names or IP addresses to route traffic to a container and if the traffic is high and multiple instances are available, Kubernetes does load balance the traffic
- **Storage orchestration**: Automatically provide storage in the form of mountable volumes
- **Automated rollouts and rollbacks**: When a new desired state is provided Kubernetes tries to achieve the state at a controlled rate and has the possibility of performing rollbacks
- **Automatic bin packing**: Kubernetes only needs to know how much CPU and RAM a workload needs and then takes care of placing the workload on a fitting node in the cluster
- **Self-healing**: If workloads are failing, Kubernetes tries to restart the applications and even kills services that do not respond to the configured health checks
- **Secret and configuration management**: Kubernetes has a store for sensitive data as well as configuration data that may change the behavior of a workload

The list of features is not complete. There are many concepts in Kubernetes that help to build complex deployment scenarios and enable teams to ship their applications in an agile manner.

Kubernetes works with containerized applications. In contrast to “plain” Docker, it orchestrates the applications and is responsible for the desired state depicted in the application manifest files. Examples of such deployments and other Kubernetes objects are available online in the documentation [1][4].

### 2.2.2 Terminology

In Table 1, we state the key terms for Kubernetes. A more complete list can be found in Appendix A in Table A.1.

Table 1: Key Kubernetes Terminology

| Term | Description |
|---|---|
| Container | The smallest possible unit in a deployment. Contains the definition of the workload. A Pod contains one or more containers. |
| Pod | Composed of multiple containers. Pod are the smalles deployable units in Kubernetes. |
| Service | A service enables (network) communication with one multiple pods. |
| CRD | A Custom Resource Definition (CRD) enables developers to extend the default Kubernetes API. |
| Operator | An operator is a software that manages Kubernetes resources and their lifecycle. Operators may use CRDs to define custom objects on which they react when some event (`Added`, `Modified` or `Deleted`) triggers on a resource. For a more in-depth description, see Section 2.3. |

[4] https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#creating-a-deployment

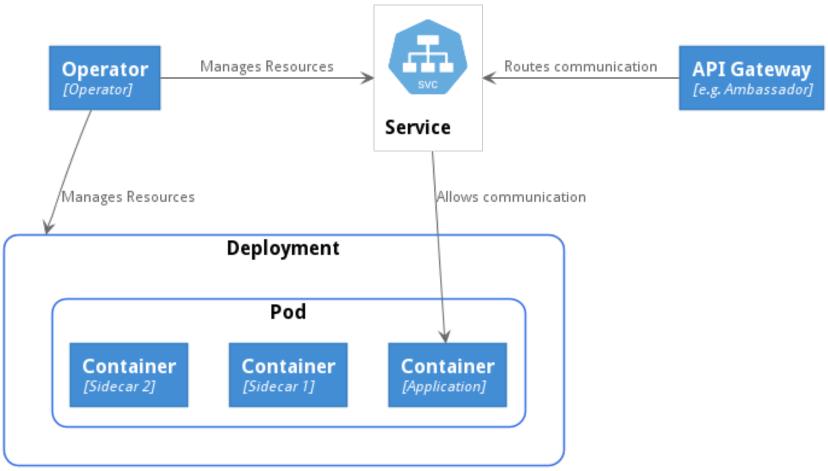

Figure 3: UML of Kubernetes Resources (partially)

Figure 3 shows a partial part of Kubernetes objects. An operator can manage resources such as deployments or services. The operator may manage other resources or custom resources based on the configuration it uses. A deployment contains a pod, which contains containers. The container is the effective unit of work, defined by an image. The service allows communication with a specific container on a configured port.

## 2.3 The Operator Pattern

An Operator can be seen as a software for Site Reliability Engineering (SRE). SRE is a set of patterns and principles that originated at Google to manage and run large applications and systems. An Operator can automatically manage a cluster of database servers or other complex applications that would require expert knowledge [3]. The term "Operator" may come from the fact, that it replaces an expert, for example a database admin, that would operate and manage the application manually.

An Operator in Kubernetes is an extension to the Kubernetes control plane and API itself. A custom Operator typically manages the whole lifecycle of an application it manages [3]. An Operator can further be used to reconcile normal Kubernetes resources or any combination thereof.

Some examples of application operators are:

- Prometheus Operator[5]: Manages instances of Prometheus (open-source monitoring and alerting toolkit) in a cluster

[5] https://github.com/prometheus-operator/prometheus-operator

- Postgres Operator[6]: Manages PostgreSQL clusters inside Kubernetes, with the support of multiple instance database clusters

There exists a broad list of operators, which can be (partially) viewed on https://operatorhub.io.

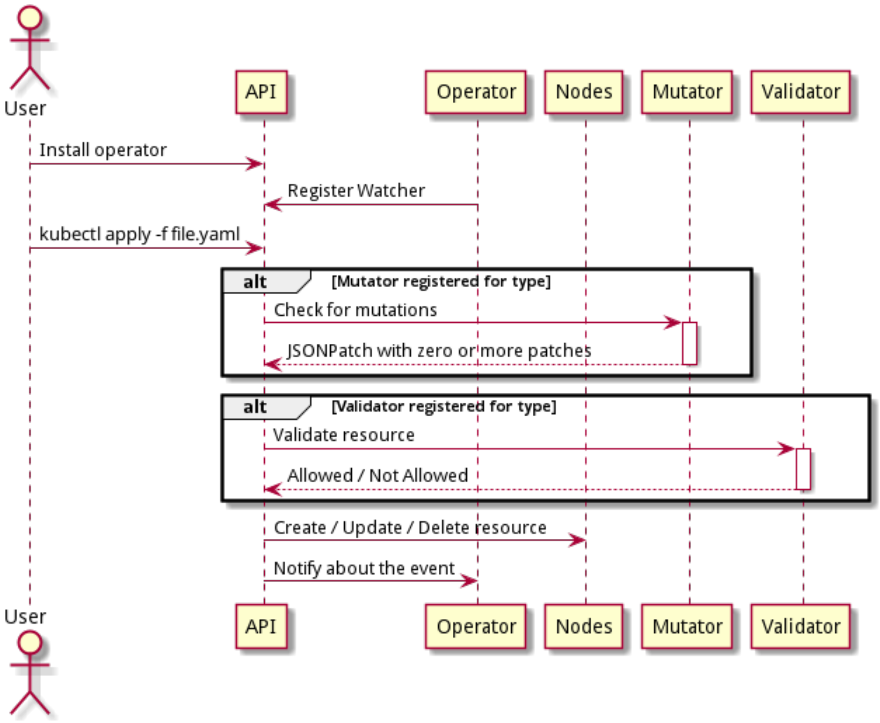

Figure 4: Kubernetes Operator Workflow

In Figure 4, we depict the general workflow of an event that is managed by an operator. When an operator is installed and runs on a Kubernetes cluster, it registers “Resource Watchers” with the API and receives notifications if the master node modifies resources. The overviewed events are “Added,” “Modified” and “Deleted.” There are two additional events that may be returned by the API (“Error” and “Bookmark”), but they are typically not needed for reconciliation.

When the user interacts with the Kubernetes API (e.g. via the `kubectl` executable) and creates a new instance of a resource, the API will first call any “Mutator” in a serial manner. After the mutators, the API will call any “Validators” in parallel and if no validator objects against the creation, the API will then store the resource and tries to apply the transition for the new desired state. Now, the operator receives a notification about the watched resource and may interact with the event. Such action may include updating resources, create more resources or even delete other instances.

[6] https://github.com/zalando/postgres-operator

A theoretical example of the concept is an Operator that creates database users based on a custom resource definition. When a user creates a custom resource with a username and a password, the Operator reacts to the creation and calls for validators to check if the username is set and the password is set. If the validation passes, the mutator may change the username according to some rules (e.g. no uppercase letters) and then the API stores the custom resource. After the resource is stored, the Operator gets notified about the effective “creation” and can reconcile the resource accordingly.

## 2.4 The Sidecar Pattern

According to Brendan Burns and David Oppenheimer, the sidecar pattern is the most common pattern for multi-container deployments [4]. Sidecars are containers that enhance the functionality of the main container in a pod. An example for such a sidecar is a log collector, that fetches log files written to the file system and forwards them towards some log processing software [4]. Another example is the Google CloudSQL Proxy[7], which provides access to a CloudSQL instance from a pod without routing the whole traffic through Kubernetes services.

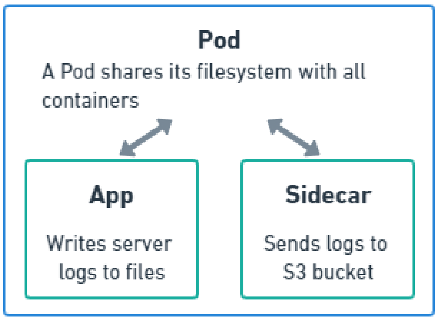
Pod
A Pod shares its filesystem with all containers
App
Writes server logs to files
Sidecar
Sends logs to S3 bucket

Figure 5: Sidecar container extending a main container in a pod. As example, this could be a log collector [4]. Both containers in the Pod share the same filesystem and can access files in the Pod. The Application writes logs into files and the Sidecar sends the logfiles into an S3 bucket.

The example shown in Figure 5 is extensible. Common use cases for sidecars include controlling the data flow in a cluster with a service mesh[8], providing access to secure locations[9] or performing additional tasks such as collecting logs of an application. Since sidecars are tightly coupled to the original application, they scale with the pod. It is not

[7] https://github.com/GoogleCloudPlatform/cloudsql-proxy
[8] As done by Istio (https://istio.io/latest/docs/reference/config/networking/sidecar/)
[9] Like the Google CloudSQL Proxy

possible to scale a sidecar without scaling the pod — and therefore the application — itself.

## 2.5 Controlling the Data with a Service Mesh

A "Service Mesh" is a dedicated infrastructure layer that handles intercommunication between services. It is responsible for the delivery of requests in a modern cloud application [5]. An example is "Istio"[10]. When using Istio, the applications do not need to know if there is a service mesh installed or not. Istio will inject a sidecar (see Section 2.4) into the deployments to handle the communication between services.

The service mesh provides a set of features [5]:

- **Service discovery**: The mechanism to locate and communicate with a workload / service. In a cloud environment, the location of services will likely change, thus, the service mesh provides a way to access the services in the cloud.
- **Load balancing**: As an addition to the service discovery, the mesh provides load balancing mechanisms as is done by Kubernetes itself.
- **Fault tolerance**: The router in a service mesh is responsible to route traffic to healthy services. If a service is unavailable or even reports a crash, traffic should not be routed to this instance.
- **Traffic monitoring**: In contrast to the default Kubernetes possibilities, with a service mesh, the traffic from and to various services can be monitored in detail. This offers the opportunity to derive reports per target, success rates and other metrics.
- **Circuit breaking**: The ability to cut off an overloaded service and back off the remaining requests instead of totally failing the service under stress. A circuit breaker pattern measures the failure rate of a service and applies states to the service: "Closed" — requests are passed to the service, "Open" — requests are not passed to this instance, "Half-Open" — only a limited number is passed [6].
- **Authentication and access control**: Through the control plane, a service mesh may define the rules of communication. It defines which services can communicate with one another.

As observed in the list above, many of the features of a service mesh are already provided by Kubernetes. Service discovery, load balancing, fault tolerance and — though limited — traffic monitoring is already possible with Kubernetes. Introducing a service mesh into a cluster enables administrators to build more complex scenarios and deployments.

[10] https://istio.io/

## 2.6 Authentication, Authorization, and Security

This section provides an introduction to the used authentication mechanisms. The proposed solution is capable of handling more than the described schemes, but for the implementation of the PoC, Basic Authentication and OIDC were used.

### 2.6.1 Basic Authentication (RFC7617)

The `Basic` authentication is a trivial authentication scheme (i.e. a way to prove the identity of an entity) that accepts a username and a password encoded in Base64. To transmit the credentials, the username and the password are concatenated with a colon (`:`) and then encoded with Base64. The result is inserted into the HTTP request as the `Authorization` header with the prefix `Basic` [7].

### 2.6.2 OpenID Connect (OIDC)

OpenID Connect is not defined in an RFC. The specification is provided by the OpenID Foundation (OIDF). OIDC extends OAuth, which is defined by **RFC6749**.

OpenID Connect is an authentication scheme, that extends/complements the `OAuth 2.0` framework. OAuth itself is an authorization framework, that enables applications to gain access to a resource (API or other) [8]. OAuth 2.0 only deals with authorization and grants access to data and features on a specific application. The OAuth framework by itself does not define *how* the credentials are transmitted and exchanged [8]. OIDC adds additional logic to OAuth 2.0 that defines *how* these credentials must be exchanged. Thus, OIDC enables login and profile capabilities in any application that uses OIDC [9].

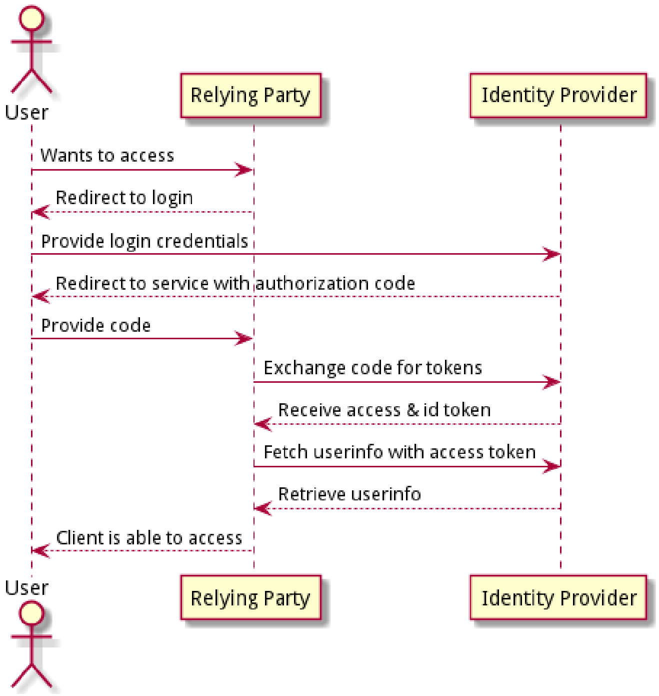

Figure 6: OIDC code authorization flow [9]. Only contains the credential flow, without the explicit OAuth part. OAuth handles the authorization whereas OIDC handles the authentication.

When a user wants to authenticate himself with OIDC, one of the possible “flows” is the “Authorization Code Flow.” Other possible flows are the “Implicit Flow” and the “Hybrid Flow” [9]. Figure 6 depicts the “Authorization Code Flow.” A user that wants to access a certain resource (e.g. an API) on a relying party (i.e. something that relies on the information about the user) and is not authenticated and authorized, the relying party forwards the user to the identity provider (IdP). The user provides his credentials to the IdP and is returned to the relying party with an authorization code. The relying party can then exchange the authorization code for valid tokens on the token endpoint of the IdP. Typically, `access_token` and `id_token` are provided. While the `id_token` must be a JSON Web Token (JWT), the `access_token` can be in any format [9].

An example of an `id_token` in JWT format may be:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
yJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.
flKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

The stated JWT token contains:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

Such JWT tokens contain information as well as a hash to secure integrity of the data. The mechanism of JWT tokens could be used to implement the “common language format” for the solution. It provides a mechanism to transmit data and protects the data against modification with a hash.

### 2.6.3 Zero Trust Environment

“Zero Trust” is a security model with a focus on protecting data and user credentials. The basic idea of zero trust is to assume that an attacker is always present. It does not matter if the application resides within an enterprise network, zero trust assumes that enterprise networks are no more trustworthy than any other public network. As a consequence of zero trust, applications are not implicitly trusted. Therefore, user credentials must be presented and validated for each access to a resource [10]. Zero trust can be summarized with: “Never trust, always verify.”

# 3 State of the Art and the Practice

This section gives an overview of the current state of the art and the practice. Furthermore, it states the deficiencies that this project tries to solve.

## 3.1 Accessing Legacy Systems from Cloud-Native Applications

In cloud environments, a solved problem is the transmission of arbitrary data from one endpoint to another. Several programming languages (like .NET, Python and Node.js) provide ways to handle communication with other endpoints and APIs. To transmit data between services in a cloud environment, an application can use the HTTP protocol, gRPC[11], or any other protocol to encode the requests and responses. In the case of a service mesh, a sidecar is injected into the pod that contains a proxy to handle data transmission between the services.

In terms of authentication and authorization, there is a variety of schemes that enable an application to authenticate and authorize its users. OpenID Connect (OIDC) is a state-of-the-art authentication scheme, that complements the OAuth 2.0 framework, which in turn handles authorization [9]. OAuth only defines how to grant access to specific resources (like APIs) but not how the access tokens are exchanged. OIDC fills that space by introducing authentication flows (e.g. “Authorization Code Flow” in Figure 6). OAuth in combination with OIDC provides a modern and secure way of authentication and authorizing users against an API.

Software architectures that are specifically designed for the cloud are called “Cloud Native Applications” (CNA). A CNA can be defined as [11]:

> “A cloud-native application is a distributed, elastic and horizontal scalable system composed of (micro)services which isolates state in a minimum of stateful components. The application and each self-contained deployment unit of that application is designed according to cloud-focused design patterns and operated on a self-service elastic platform.”

However, with CNAs and the general movement to cloud environments and digitalization, not all applications get that chance to adjust. For various reasons like budget, time or technical risks and skill availability, legacy applications and monoliths are not always refactored or re-written before they are deployed into a cloud environment. Michael Feathers wrote an impressive book on how to work with legacy code. With the guidance of the book, it may be possible for some applications to be modernized [12].

If legacy applications (for example an old enterprise resource planning system) are mixed with CNAs, then the need of “translation” arises. Assuming that the CNA is a secure application that uses OIDC to authenticate its users and the application needs

[11] https://grpc.io/

to fetch data from a legacy system. The legacy application does not understand OIDC, thus, either the modern or the legacy application must receive code changes (i.e. enable the application to convert the user credentials to the scheme of the target service) to enable communication between the services. Following the previous assumption, the code changes will likely be introduced into the CNA, since it is presumably better maintainable and deployable than the legacy app. Hence, the modern application receives changes that may introduce new bugs or vulnerabilities. If new code is introduced into an application, “normal” software bugs may be created and external dependencies (such as libraries for authentication and authorization) may import vulnerabilities caused by bugs or by deviation from standards.

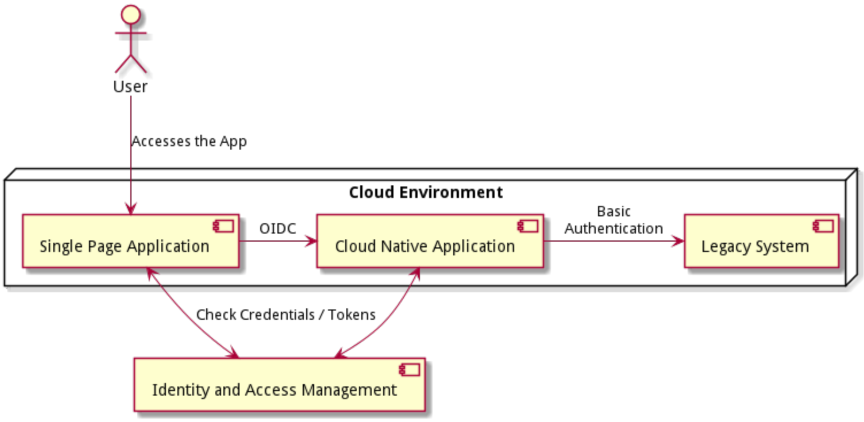

Figure 7: Microservice architecture that contains modern applications as well as legacy services.

We consider the components in Figure 7:

- **User**: A person with access to the application
- **Single Page Application**: A modern single page application (SPA)
- **Identity and Access Management (IAM)**: Identity Provider for the solution (does not necessarily reside in the same cloud)
- **Cloud Native Application (CNA)**: A state-of-the-art API application and primary access point for the client
- **Legacy System**: Legacy service that is called by the CNA to fetch some additional data

In practice, we encountered the stated scenario at various points in time. Legacy services may not be the primary use case. Another example is the usage of third-party applications without any access to the source code.

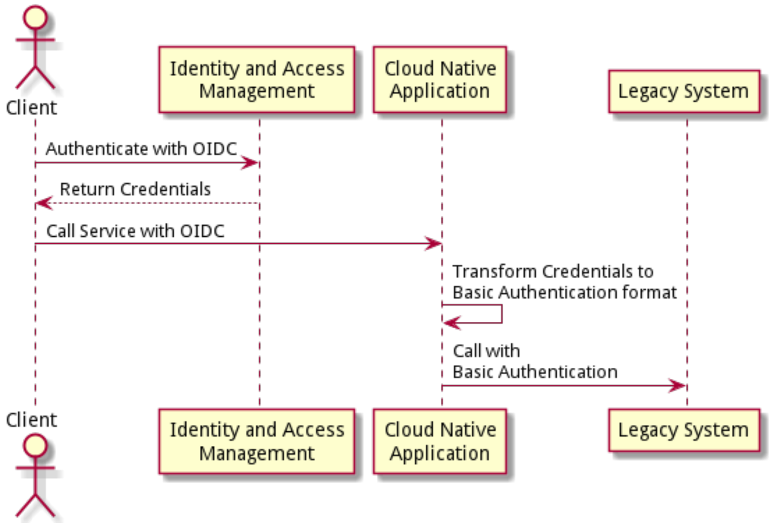

Figure 8: Current state of the art of accessing legacy systems from modern services with differing authentication schemes.

The invocation sequence in Figure 8 shows the process of communication in such a scenario. In Figure 8, the SPA (Client) authenticates against an arbitrary IAM. The CNA is the modern backend that supports the SPA as a backend API. Therefore, the CNA provides functionality for the SPA. The legacy application, for example an old ERP with order information, was moved into the cloud, but is not refactored nor re-written to communicate with modern authentication technologies.

In this scenario, the SPA calls some API on the CNA that — in turn — will call the legacy system to get additional information to present to the user. Since the SPA and the CNA communicate with the same authentication technology, the call is straightforward. The SPA authenticates itself and obtains an access token. When calling the service (the CNA), the token is transmitted and the service can check with the IAM if the user is authorized to access the system. When the CNA calls the legacy system for additional information, it is required to translate the user-provided credentials (the access token) to a format that the legacy system understands. In the example, the legacy system is only able to handle Basic Authentication (RFC7617). This means, if the CNA wants to communicate with the legacy system, it must implement some translation logic to change the user credentials into the typical Basic Authentication Base64 encoded format of `<Username>:<Password>`. Hence, code changes are introduced to the CNA since the legacy system is not likely to be easily maintainable.

## 3.2 External Authentication and Identity Transport

In practice, no current solution exists that allows credentials to be transformed between authentication schemes. The service mesh “Istio” provides a mechanism to secure services that communicate with mTLS (mutual TLS) [13] as well as an external mechanism to provide custom authentication and authorization capabilities [14]. The concept of Istio works well when all applications in the system share the same authentication scheme. As soon as two or more schemes are in place, the need for transformation arises again.

In fact, the external authentication feature of Istio is based on Envoy. Istio uses Envoy as an injected sidecar. Another prominent API gateway, “NGINX”[12], implements a similar external authentication mechanism [15]. However, Envoy implements a more fine-grained control over the HTTP request. As an external authentication service for Envoy, the result may change HTTP headers in the request and the response. While both gateways offer a way of external authentication and authorization, they cannot transform the user credentials on their own. In Envoy, the external service may attach or modify HTTP headers, while in NGINX, the external service may only allow or reject a request.

There exist techniques, such as SAML (Security Assertion Markup Language) or JWT (JSON web token) profiles, to transmit an identity of a user to other services. However, SAML only describes the format of the identity itself, not the translation between varying types of credentials. SAML works when all participating services understand SAML as well. If a legacy system is not able to parse and understand SAML, the same problem arises.

All the discussed technologies and applications above do not support the dynamic conversion of user credentials. While Istio solves the communication and enables mTLS between services, it does not translate credentials between services. SAML gives a common format for an identity of a user, but it is an authentication scheme on its own. Thus, the “translation problem” still persists.

## 3.3 Missing Dynamic Credential Transformation

The situation described in the previous sections introduces several problems. It does not matter whether the legacy system is a third-party application to which no code changes can be applied to, or if it is an application that cannot be updated for the time being. Most likely, the code change to provide the ability to communicate will be introduced into the CNA. This adds the risk of errors since new code must be produced, which would not be necessary if the legacy service was refactored. Also, changing the CNA to communicate with the legacy software may be a feasible solution in a small setup. But as the landscape of the application grows, this solution does not scale well.

[12] https://www.nginx.com/

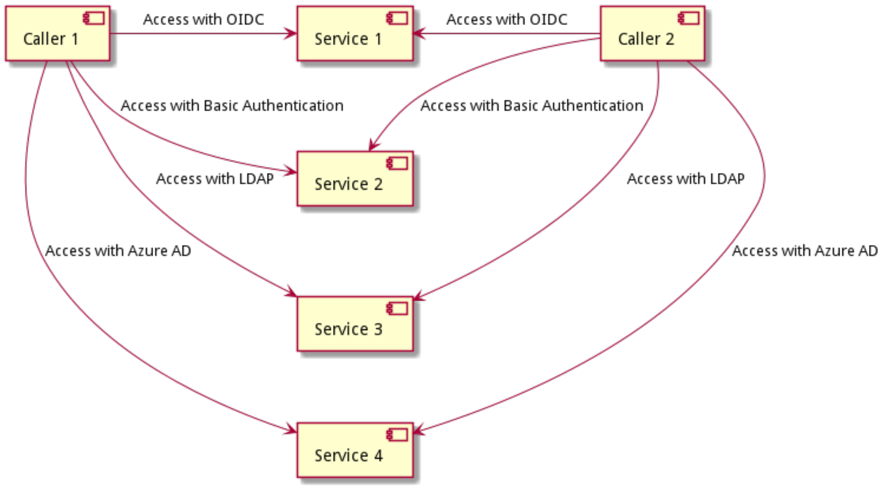

Figure 9: Service Landscape with various authentication mechanism

The problem, as depicted in Figure 9, shows that the number of conversion mechanisms increases with each service and each authentication method. As the landscape and the different methods of authentication grow, it is not a feasible solution to implement every authentication scheme in all the callers. In Figure 9, “Caller 1” is required to transform the user credentials into four different formats to communicate with services one to four. When another caller enters the landscape, it must implement the same four mechanisms as well.

Another issue that emerges with this transformation of credentials: The credentials leak into the trust zone. As long as each service is in the same trust zone (for example, in the same data center in the same cluster behind the same API gateway), this may not be problematic. As soon as the communication is between data centers, the communication and the credentials must be protected. It is not possible to create a zero trust (the assumption, that an attacker is always present) environment with the need of knowledge about the target’s authentication schemes.

Service meshes may provide a way to secure communication between services, but they are not able to transform credentials to a required format for any legacy application yet. It would be a possible solution to enable service meshes to transform credentials. However, service meshes introduce another layer of complexity on top of the environment. Such a system should be easy to use.

Other technologies — such as credential vaults — have a similar problem. The vault is the central weakness in the system. If the vault is attacked, the whole trust zone may fail. While a credential vault would provide a way to share credentials between services, it does not mitigate the need of transformation. A vault, like “Vault by HashiCorp”[13], typically

[13] https://www.vaultproject.io/

provides a secure way to inject credentials into a system. The vaults do not transform credentials for the destination. However, such a vault could be used as secure storage of credentials for such a system that enables the described transformation. In the example of Figure 8, the secure vault could be used to store the static Basic Authentication credentials for the legacy service. The transformer can then access the vault to fetch the needed credentials for the target system.

# 4 Distributed Authentication Mesh

This section gives an overview and an in-depth documentation of the proposed solution. Furthermore, boundaries of the solution are provided along with common software engineering elements like requirements, non-functional requirements, an abstract and a conceptional architecture.

The proposed architecture provides a generic description for a solution to the described problem. For this project, a Proof of Concept (PoC) gives insights into the topic of manipulating HTTP requests in-flight. The PoC is implemented to run on a Kubernetes cluster to provide a practical example.

## 4.1 Definition

A solution for the stated problems in Section 3 must be stateless and able to transform arbitrary credentials into a format that the target service understands. For this purpose, the architecture contains a service that runs as a sidecar among the target service. This sidecar intercepts requests to the target and transforms the Authorization HTTP header. The sidecar is used to intercept inbound and outbound traffic.

However, the solution **must not** interfere with the data flow itself. The problem of proxying data from point A to B is well solved. In the given work, an Envoy proxy delivers data between the services. Envoy allows the usage of an external service to modify requests in-flight.

## 4.2 Goals and Non-Goals of the Project

This section presents the functional and non-functional requirements and goals for the solution. It is important to note that the implemented Proof of Concept (PoC) will not achieve all goals. Further work is needed to implement a solution according to the architecture that adheres to the stated requirements.

In Table 2, we present the list of functional requirements or goals (REQ) for the proposed solution and the project in general.

Table 2: Functional Requirements

| Name | Description |
|---|---|
| REQ 1 | The translator module must be able to transform given credentials into the specified common language and the common format back into valid credentials. |
| REQ 2 | The translator handles errors if they occur. When an unrecoverable error happens, the request is rejected. |

| Name | Description |
|---|---|
| REQ 3 | A proxy is deployed to intercept communication with the service in question to handle the data flow. |
| REQ 4 | Translators do only modify HTTP headers. They must not interfere with the data that is transmitted. |
| REQ 5 | The automation engine decides which elements are relevant for the authentication mesh. |
| REQ 6 | The automation engine — if it exists — enhances objects with the proxy and translator engine. |

In Table 3, we show the non-functional requirements or non-goals (NFR) for the proposed solution.

Table 3: Non-Functional Requirements

| Name | Description |
|---|---|
| NFR 1 | First and foremost, the solution **must not** be less secure than current solutions. |
| NFR 2 | The solution must adhere to current best practices and security mechanisms. Furthermore, it **must** be implemented according to the standards of the practice to mitigate security issues as stated in the OWASP Top Ten (https://owasp.org/www-project-top-ten). |
| NFR 3 | The concept of the solution is applicable to cluster orchestration software other than Kubernetes. The architecture provides a general way of solving the stated problem instead of giving a proprietary solution for one vendor. The concept should even be realizable for non-orchestration platforms like a Windows operating system. |
| NFR 4 | The translation of the credentials should not extensively impact the timeframe of an arbitrary request. In production mode, the additional time to check and transform the credentials **should** not exceed 100ms. This is a general recommendation and some authentication mechanism may exceed the stated 100ms. |
| NFR 5 | The solution must be extensible with additional "translators" that provide the means of transforming the given credentials to other target formats. |
| NFR 6 | The solution may run with or without a service mesh. It is a goal that the solution can run without a service mesh to reduce the overall complexity, but if a service mesh is already in place, the solution must be able to work with the provided infrastructure. |
| NFR 7 | The architecture **must** be scalable. In a cloud-native environment, the application that is enhanced may be scaled. Therefore, the solution must be able to scale with the application as well. |

| Name | Description |
|---|---|
| NFR 8 | Each translator **should** only handle one authentication scheme to ensure separation of concerns and scalability of the whole solution. |
| NFR 9 | The solution depends on an external software for data transmission. The solution **must not** interfere with the data plane. Error handling of the data plane is handled by the external application. |
| NFR 10 | The solution handles errors in the translation and the automation engine according to the architectural description. |

These goals and non-goals define the first list of REQ and NFR. During future work, this list may change to adjust to new challenges.

## 4.3 Differentiation from Security Assertion Markup Language

The “Security Assertion Markup Language” (SAML) is a so-called “Federated Identity Management” (FIdM) standard. SAML, OAuth, and OIDC represent the three most popular FIdM standards. SAML is an XML framework for transmitting user data, such as authentication, entitlement, and other attributes, between services and organizations [16].

While SAML is a partial solution for the stated problem, it does not cover the use case when credentials need to be transformed to communicate with a legacy system. SAML enables services to share identities in a trustful way, but all communicating partners must implement the SAML protocol to be part of the network. This project addresses the specific transformation of credentials into a format for some legacy systems. The basic concept of SAML may be used as a baseline of security and the general idea of processing identities.

## 4.4 Architecture of the Distributed Authentication Mesh

The following sections provide an architectural description of the proposed solution. First, a description gives an initial overview of the architecture and the conceptional idea. Afterward, an abstract architecture describes the concepts behind the distributed authentication mesh. Then the architecture is concretized with platform-specific examples based on Kubernetes.

The reader should note that the proposed architecture does not match the implementation of the PoC to the full extent. The goal of this project is to provide an abstract idea to implement such an authentication mesh, while the PoC proves the ability to modify HTTP requests in-flight.

### 4.4.1 Federated Identity with Diverging Authentication Schemes

When a federated identity is used, a user is not required to present authentication credentials for each communication between services. At some point, the user validates his own identity and is authenticated in the application. This application can span over several services that share the same “trust.” This does not contradict a zero-trust environment. A federated identity can be validated by each service and thus may be used in a zero-trust environment.

To achieve such a federated identity with diverging authentication schemes, the solution converts validated credentials (like access tokens) to a domain specific language (DSL). This format, in conjunction with a proof of the sender, validates the identity over the wire in the communication between services without the need of additional authentication. When all parties of a communication are trusted through verification, no information about the effective credentials leaks into the communication between services.

The concept of the distributed authentication mesh is to replace any user credentials from an outgoing HTTP request with the DSL representation of the user identity. On the receiving side, the DSL encoded identity in the incoming HTTP request is transformed to the valid user credentials for the target service.

Since the topic of the mesh is security, error handling is a delicate matter. The mesh does depend on existing infrastructure and principles. In the example of the PoC, that is implemented on Kubernetes, error handling relies on Kubernetes. The Operator injects the translators and proxies and Kubernetes is responsible for the operational state of those components. Thus, error handling is limited to the translator engine, which represents the critical element in the solution. When the translator encounters any error and the translator can not recover from the error, the request must be denied. The translator may crash, and it lies in the responsibility of Kubernetes to restart the translator.

### 4.4.2 Conceptional Architecture

This section describes the architecture of the proposed solution in an abstract and generalized way. As stated in the non-functional requirements, the concepts are not bound to any specific platform or a specific implementation nor required to run in a cloud environment. The concepts could be implemented as a “fat-client” solution for a Windows machine.

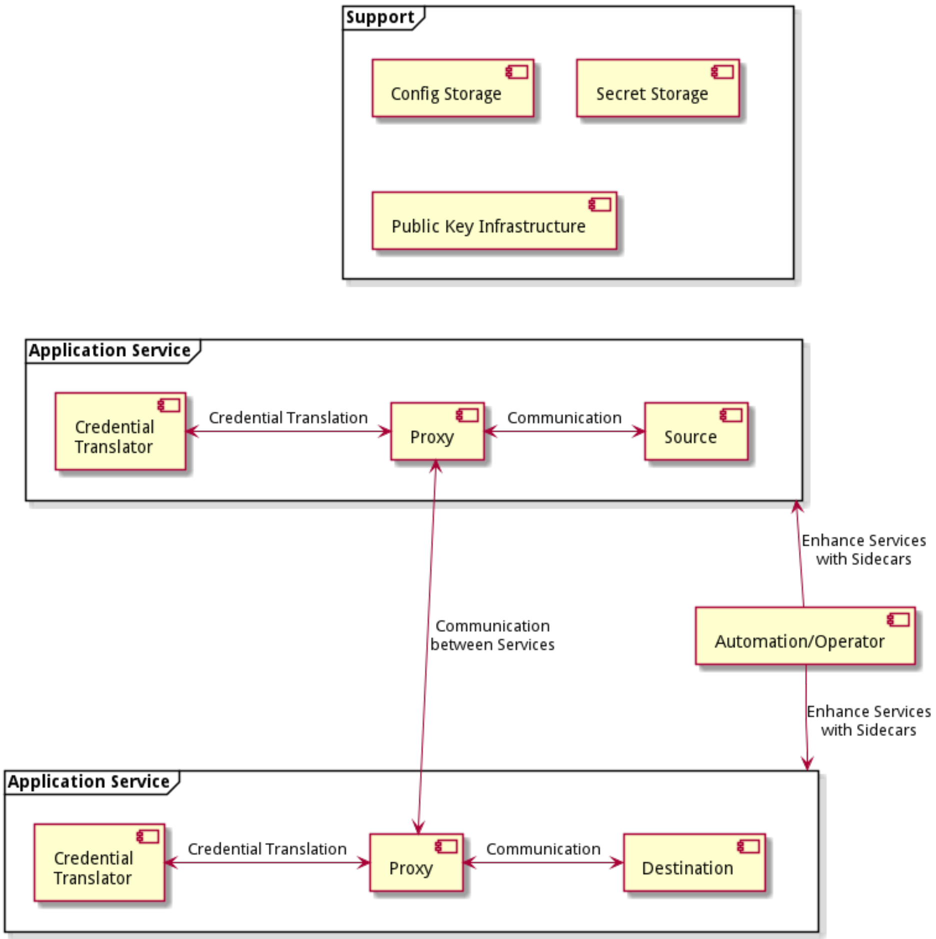

Figure 10: Abstract Solution Architecture

Figure 10 shows the abstract solution architecture. In the "support" package, generally available elements provide utility functions to the mesh. The solution requires a public key infrastructure (PKI) to deliver key material for signing and validation purposes. This key material may also be used to secure the communication between the nodes (or applications). Configuration and secret storage enable the applications to store and retrieve configurations and secret elements like passwords or key material.

Additionally, an optional automation component watches and manages applications. This component enhances the application services with the required components to participate in the distributed authentication mesh. Such a component is strongly suggested when the solution is used in a cloud environment to enable dynamic usage of the mesh. The automation injects the proxies, translators, and the required configurations for the managed components.

A (managed) application service consists of three parts. The source (or destination) service, which represents the deployed application itself, a translator that manages the transformation between the DSL of the identity and the implementation specific authentication format, and a proxy that manages the communication from and to the application.

The communication between instances in the authentication mesh is handled by the proxies. The mesh must not interfere with the data transmission, it is only responsible for modifying HTTP headers. Handling errors on the data plane is not part of the mesh and must be done by the implementation of the proxy.

### 4.4.3 Platform-Specific Example in Kubernetes

For the following sections, the architecture shows elements of a Kubernetes cloud environment. The reason is to describe the specific architecture the context of the practice. Table A.1 explains used terms and concepts in Kubernetes which are used to describe the platform-specific architecture.

Since the example is Kubernetes specific, error handling and recovery mechanisms of Kubernetes can be used. So if a part of the mesh crashes due to an unexpected error, Kubernetes is responsible for restarting that part. Furthermore, Kubernetes is the orchestrator which takes actions to provide the running state of all applications. If any errors are encountered, proper logging must be provided.

**4.4.3.1 Automation with an Operator** The automation part of the mesh is optional. When no automation is provided, the required proxy and translator elements must be started and maintained by some other means. However, in the context of Kubernetes, an Operator pattern enables an automated enhancement and management of applications.

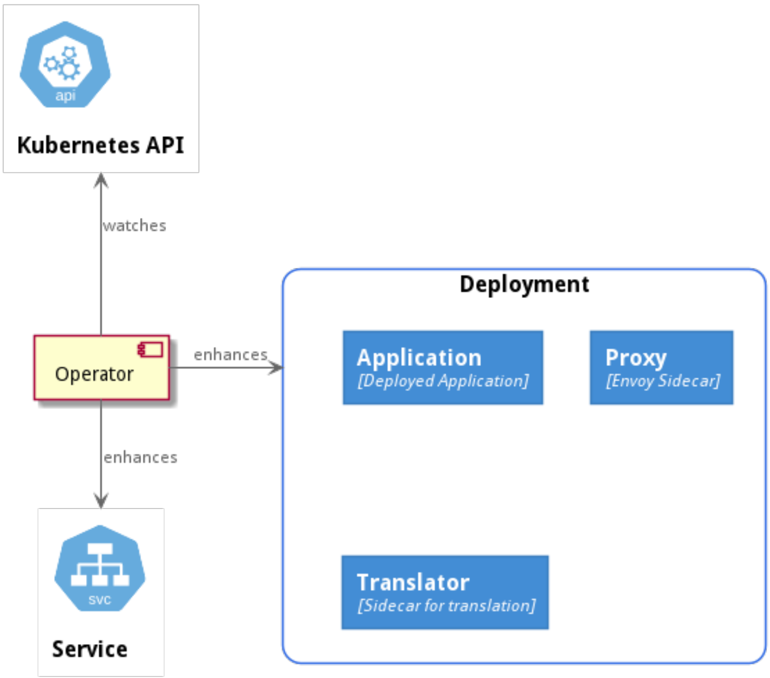

Figure 11: Automation with an Operator in a Kubernetes Environment

The Operator (application lifecycle manager, see Section 2) in Figure 11 watches the Kubernetes API for changes. When deployments or services are created, the Operator enhances the respective elements. “Enhancing” means that additional containers are injected into a deployment as sidecars. The additional containers contain the proxy and the translator. While the proxy manages incoming and outgoing communication, the translator manages the transformation of credentials from and to the DSL.

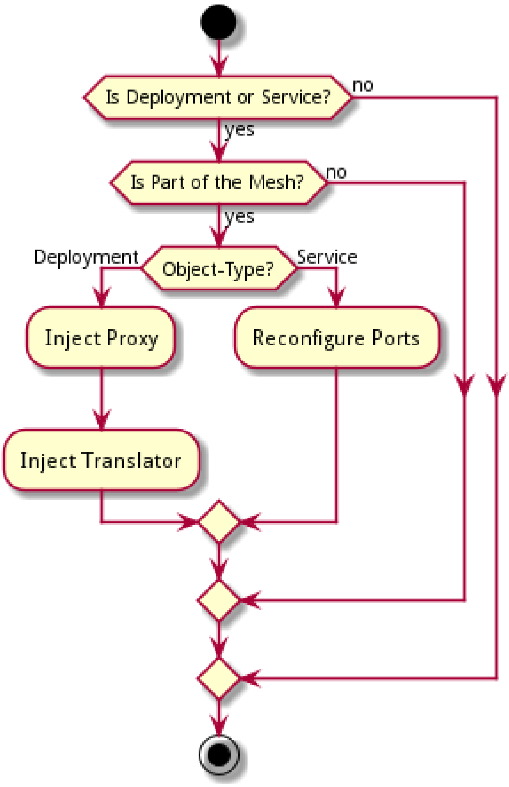

Figure 12: The Operator determines the relevance of an object with this logic. If an object in Kubernetes is not a Deployment nor a Service, or does not contain specific “Labels,” it is rejected.

To determine if an object is relevant for the automation, the operator uses the logic shown in Figure 12. If the object in question is not a deployment (or any other deployable resource, like a “Stateful Set” or “Daemon Set”) or a service, then it is not relevant for the mesh. If the object is not configured to be part of the mesh, then the automation ends here as well.

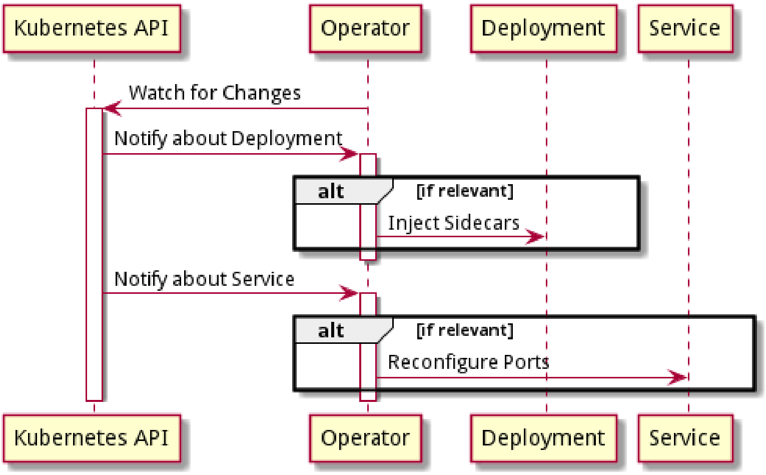

Figure 13: Automated Enhancement of a Deployment and a Service. If the Operator decides that an object is relevant (see Figure 12), the object is enhanced depending on its type.

The sequence that enhances deployments and services is shown in Figure 13. The operator registers a “watcher” for deployments and services with the Kubernetes API. Whenever a deployment or a service is created or modified, the operator receives a notification. Then, the operator checks if the object in question “is relevant” by checking if it should be part of the authentication mesh. This participation can be configured — in the example of Kubernetes — via annotations, labels, or any other means of configuration. If the object is relevant, the operator injects sidecars into the deployment or reconfigures the service to use the injected proxy as the target for the network communication.

If the automation engine encounters errors, it relies on Kubernetes to perform actions to reach a meaningful state. Since the engine runs on Kubernetes, if any operational errors occur, the application is restarted by Kubernetes. Logging is essential to find such errors. If deployments and services cannot be modified, the operator shall try again in the next reconciliation cycle.

**4.4.3.2 Public Key Infrastructure** The role of the public key infrastructure (PKI) in the solution is to act as the source for trust in the system. The PKI is responsible for generating and delivering key material to various components. As an example, a translator fetches a public/private key pair on startup and can sign the translated credentials with the key material. A receiver can then validate the signature and check the integrity of the transmitted data.

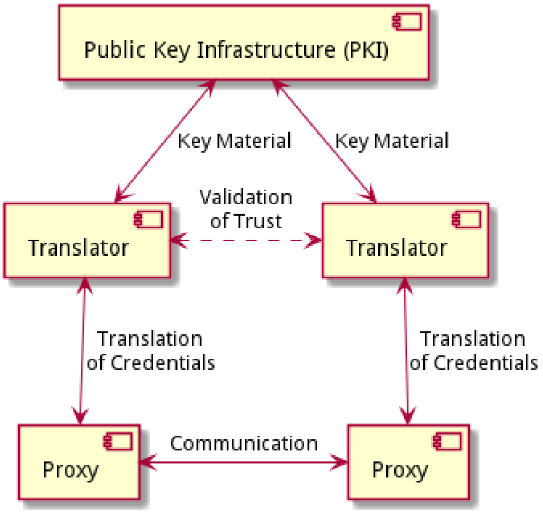

Figure 14: The Relation of the Public Key Infrastructure and the System

Figure 14 depicts the relation of the translators and the PKI. When a translator starts, it acquires trusted key material from the PKI (for example, with a certificate signing request). This key material provides the possibility to sign the identity that is transmitted to the receiving party. The receiving translator can validate the signature of the identity and the sending party. The proxies are responsible for the communication between the instances.

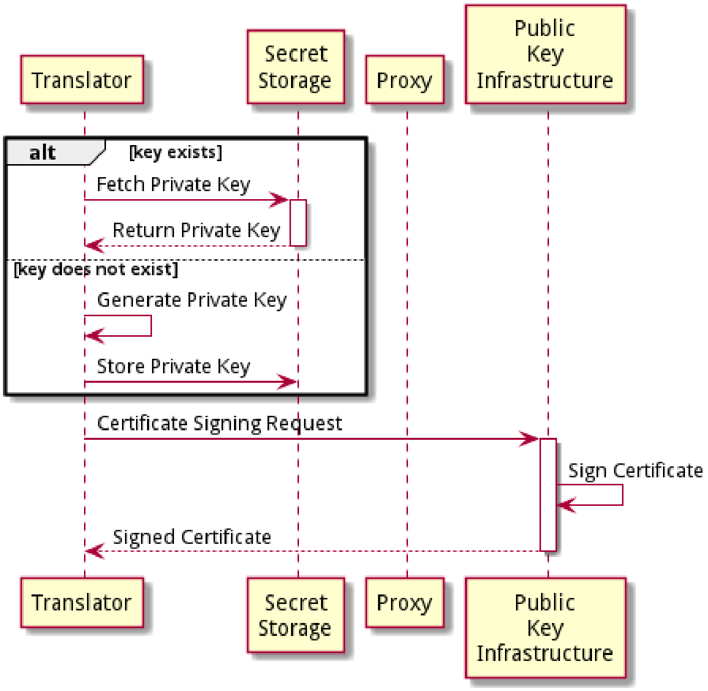

Figure 15: Provide Key Material to the Translator

The sequence in Figure 15 shows how the PKI is used by the translator to create key material for itself. When a translator starts, it checks if it already generated a private key and obtains the key (either by creating a new one or fetching the existing one). Then, a certificate signing request (CSR) is sent to the PKI. The PKI will then create a certificate with the CSR and return the signed certificate. The provided sequence shows one possible use case for the PKI. During future work, the PKI may also be used to secure communication between proxies with mTLS [17].

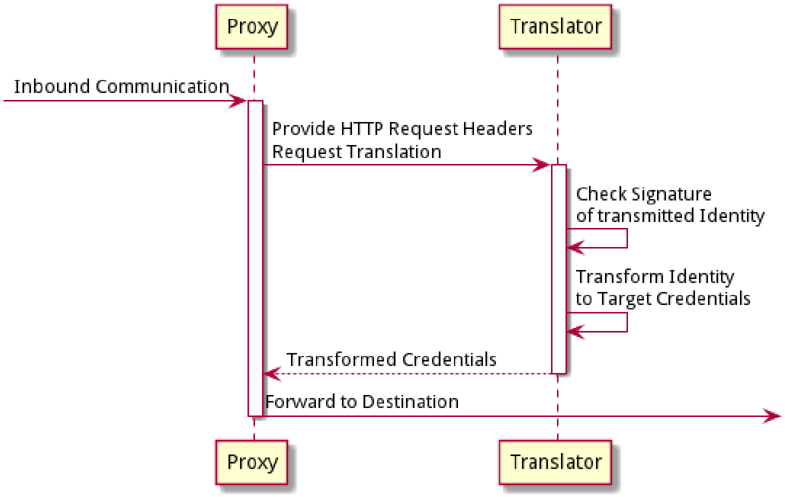

Figure 16: Checking the Signature of the transmitted Identity

When communication happens, as shown in Figure 16, the proxy forwards the HTTP headers that contain the transferred identity of the user in the DSL to the translator. In the case of a JWT token, the transformer may now confirm the signature of the JWT token with the obtained certificate since it is signed by the same Certificate Authority (CA). Then the transformation is performed and the proxy forwards the communication to the destination.

To increase the security and mitigate the problem of leaking certificates, it is advised to create short-living certificates in the PKI and refresh certificates periodically.

If the PKI encounters illegal signing requests, it must deny them. If any other unexpected errors happen, the application should log the error and then crashes to enable Kubernetes to restart the application again.

**4.4.3.3 Networking with a Proxy** Networking in the proposed solution works with a combination of routing and communication proxying. The general purpose of the networking element is to manage data transport between instances of the authentication mesh and route the traffic to the source/destination.

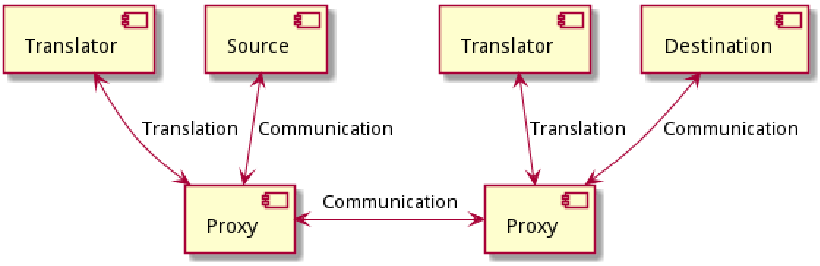

Figure 17: Networking with an Proxy

As seen in Figure 17 the proxy is the mediator between source and destination of a communication. Additionally, he proxy manages the translation of the credentials by communicating with the translator to transform the identity of the authenticated user and transmit it to the destination where it gets transformed again. In addition, with the help of the PKI, the proxy can verify the identity of the sender via mTLS.

Since the authentication mesh relies on external software to take care of communication and networking, error handling is off-loaded to that specific software as well. The authentication mesh does not guarantee any connectivity between parts of the mesh. In the platform-specific example, if the configuration provided by the automation engine is faulty, Envoy will crash and log this matter to the standard output (i.e. the console). Any other errors encountered by Envoy result in their respective HTTP error messages.

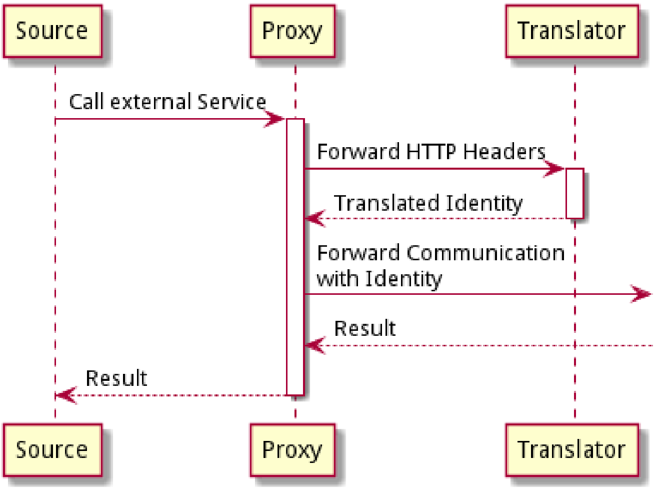

Figure 18: Outbound Networking Sequence

**Outbound Communication for an Application** In Figure 18 the outbound traffic flow is shown. The proxy is required to catch all outbound traffic from the source and performs the reversed process of Figure 19 by transforming the provided credentials from the source to generate the common format with the user identity. This identity is then inserted into the HTTP headers and sent to the destination. At the sink, the process of Figure 19 takes place — if the sink is part of the authentication mesh.

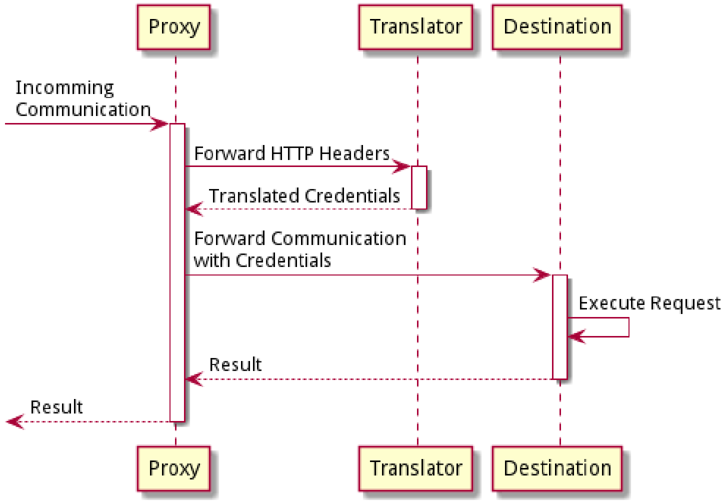

Figure 19: Inbound Accepted Networking Sequence

**Inbound Accepted Communication for an Application** Figure 19 shows the general invocation during inbound request processing. When the proxy receives a request (in the stated example by the configured Kubernetes service), it calls the translator with the HTTP request detail. The PoC is implemented with an "Envoy" proxy. Envoy allows an external service to perform "external authorization"[14] during which the external service may:

- Add new headers before reaching the destination
- Overwrite headers before reaching the destination
- Remove headers before reaching the destination
- Add new headers before returning the result to the caller
- Overwrite headers before returning the result to the caller

The translator uses this concept to consume a specific and well-known header to read the identity of the authorized user in the DSL. The identity is then validated and transformed to the authentication credentials needed by the destination. Then, the translator instructs Envoy to set the credentials for the upstream. In the PoC, this is achieved by setting the `Authorization` header to static Basic Authentication (RFC7617) credentials.

[14] https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/ext_authz_filter

**Inbound rejected Communication for an Application** If the incoming communication contains faulty, invalid, or no identification data, the proxy blocks the communication.

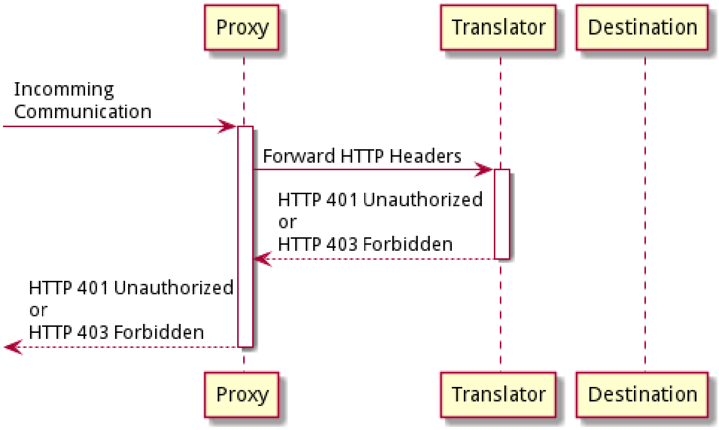

Figure 20: Inbound Rejected Networking Sequence

Figure 20 shows the sequence when no or invalid identity data is provided. The responses of the translator are defined in **RFC1945** and are the HTTP response status codes [18]. The translator distinguishes two cases:

- No identity data
- Invalid identity data

If no identity data is present, the translator must return `HTTP 401 Unauthorized`, the error that is used when no authorization credentials are provided. When invalid authorization credentials are provided (a false or a modified identity), the translator must return `HTTP 403 Forbidden`, which is used when credentials are provided, but they are not valid [18].

**4.4.3.4 The Translation of Credentials to an Identity** The translator is responsible for transforming the identity from and to the domain-specific language (the common format). In conjunction with the PKI, the translator can verify the validity and integrity of the incoming identity.

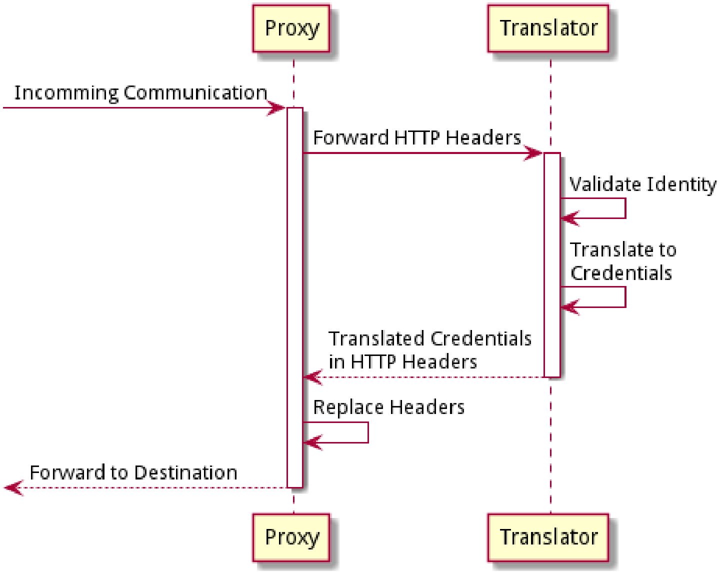

Figure 21: Translation of the transmitted user identity from the common format to the required format by the destination

When the translator receives a request to create the required credentials, it performs the sequence of actions as stated in Figure 21. First, the proxy will forward the HTTP request data to the translator. Afterward, the translator checks if the transported identity is valid and signed by an authorized party in the authentication mesh. When the credentials are valid, they are translated according to the implementation of the translator. The proxy is then instructed with the actions to replace the transported identity with the correct credentials to access the destination.

The translator is the critical part of the authentication mesh. If it receives invalid credentials (e.g. an identity that has been tampered with, or just a wrong username/password combination), it must reject the request with a `HTTP 403 Forbidden` response. If no identity is provided at all, a `HTTP 401 Unauthorized` must be sent. When the translation engine encounters any unexpected error during translation of the identity (like not being able to access the secret storage, or failure of some database), it must reject the request. The translator must reject any request that cannot be transformed successfully. This error handling is used on the receiving and the sending side.

In the PoC, the proof of integrity is not implemented, but the transformation takes place, where a "Bearer Token"[15] is used to check if the user may access the destination and then replaces the token with static Basic Authentication credentials.

[15] Access token of an IDP.

## 4.5 Securing the Communication between Applications

The communication between the proxies must be secured. Furthermore, the identity that is transformed over the wire must be tamper-proof. Two established formats would suffice, “SAML” and “JWT Tokens.” While both provide the possibility to hash their contents and thus secure them against modification, in current OIDC environments, JWT tokens are already used as access and/or identity tokens. JWT provides a secure environment with public and private claim names [19].

Other options to encode the identity are:

- Normal JSON
- YAML
- XML
- X509 Certificates
- Concise Binary Object Representation (CBOR) [20]

The problem with other structured formats is that tamper protection and encoding must be implemented manually. JWT tokens provide a specified way of attaching a hashed version of the complete content and therefore provide a method of validating a JWT token if it is still pristine and if the sender is trusted [19]. If the receiving end fetched the key material from the same PKI (and therefore the same CA), it can check the certificate and the integrity of the JWT token. If the signature is correct, the JWT token has been issued by a trusted and registered instance of the authentication network.

X509 certificates — as defined in **RFC5280** [21] — introduce another valid way of transporting data and attributes to another party. “Certificate Extensions” can be defined by “private communities” and are attached to the certificate itself [21].

While X509 certificates could be used instead of JWT to transport this data, using certificates would enforce the translator to act as intermediate CA and create new certificates for each request. From our experience, creating, extracting, and manipulating certificates, for example in C#, is not a task done lightly. Since this solution should be as easy to use as it can be, manipulating certificates in translators do not seem to be a feasible option. For the sake of simplicity and well-known usage, further work on this project will probably use JWT tokens to transmit the identity data.

## 4.6 Implementation Proof of Concept (PoC)

To prove that the general idea of the solution is possible, a PoC is implemented during the work of this project. The following technologies and environments build the foundation of the PoC:

- Environment: The PoC is implemented on a Kubernetes environment to enable automation and easy deployment for testing

- "Automation": A Kubernetes operator, written in .NET (C#) with the "Dotnet Operator SDK"[16]
- "Proxy": Envoy proxy which gets the required configuration injected as Kubernetes ConfigMap file
- "Translator": A .NET (F#) application that uses the Envoy gRPC definitions to react to Envoy's requests and poses as the external service for the external authorization
- "Sample Application": A solution of three applications that pose as demo case with:
  - "Frontend": An ASP.NET static site application that authenticates itself against "ZITADEL"[17]
  - "Modern Service": An ASP.NET API application that can verify an OIDC token from ZITADEL
  - "Legacy Service": A "legacy" ASP.NET API application that is only able to verify `Basic Auth` (RFC7617, see Section 2.6.1)

The PoC addresses the following questions:

- Is it possible to intercept HTTP requests to an arbitrary service
- Is it further possible to modify the HTTP headers of the request
- Can a sidecar service transform given credentials from one format to another
- Can a custom operator inject the following elements:
  - The correct configuration for Envoy to use external authentication
  - The translator module to transform the credentials

Based on the results of the PoC, the following further work may be possible:

- Specify the concrete format to transport identities
- Implement a secure way of transporting identities with validation of the integrity
- Provide production-ready versions for some translators and the operator
- Integrate the solution with a service mesh
- Further investigate the possibility of hardening the communication between services (e.g. with mTLS)

For the solution to be production-ready, at least the secure communication channel between elements of the mesh as well as the DSL for the identity must be implemented. To be of use in current cloud environments, an implementation in Kubernetes can provide insights on how to develop the solution for other orchestrators than Kubernetes.

When considering the abstract architecture in Figure 10, the PoC on Kubernetes covers all elements but the PKI. The automation engine is implemented with a custom Operator as stated above. The proxy is a configured Envoy proxy configured by the Operator.

[16] https://github.com/buehler/dotnet-operator-sdk

[17] https://zitadel.ch

The credential transformer is a custom software written in .NET (F#). Config and Secret Storage are covered by Kubernetes itself with “ConfigMap” and “Secret” objects in Kubernetes.

### 4.6.1 Case Study for the PoC

The demo application demonstrates the particular use case of the distributed authentication mesh. The application resides in an open-source repository on GitHub (https://github.com/WirePact/poc-showcase-app).

To install and run the case study without any interference of the Operator or the rest of the solution, follow the installation guide in the README on https://github.com/WirePact/poc-showcase-app. To install and use the whole PoC, following the instructions in Appendix B will install the operator and the case study.

When installed in a Kubernetes cluster, a user can open (depending on the local configuration) the URL to the frontend application[18].

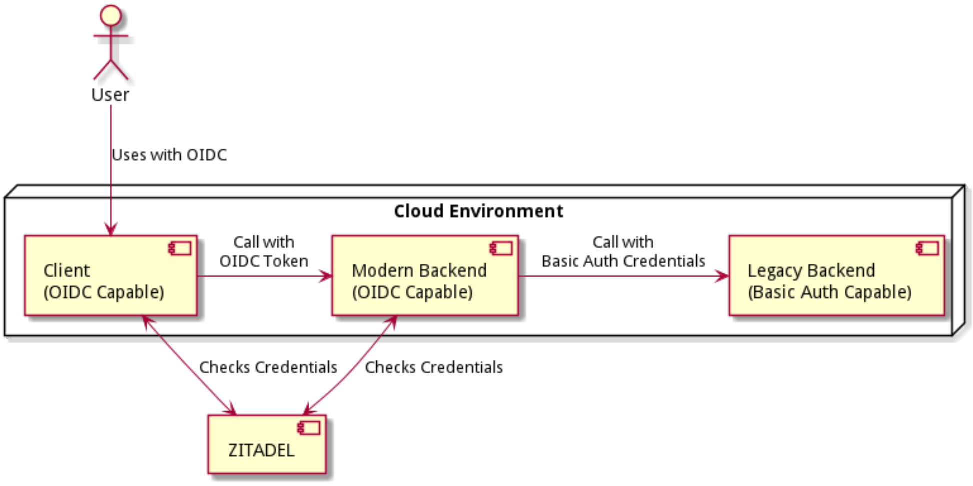

Figure 22: Component Diagram of the Case Study

Figure 22 gives an overview of the components in the showcase application. The system contains an ASP.NET Razor Page[19] application as the frontend, an ASP.NET API application with configured ZITADEL OIDC authentication as “modern” backend service, and another ASP.NET API application that only supports Basic Authentication as “legacy” backend. The frontend can only communicate with the modern API while the modern API can call an additional service on the legacy API.

[18] In the example, it is “https://kubernetes.docker.internal” since this is the local configured default URL for “Docker Desktop”

[19] https://docs.microsoft.com/en-us/aspnet/core/razor-pages/

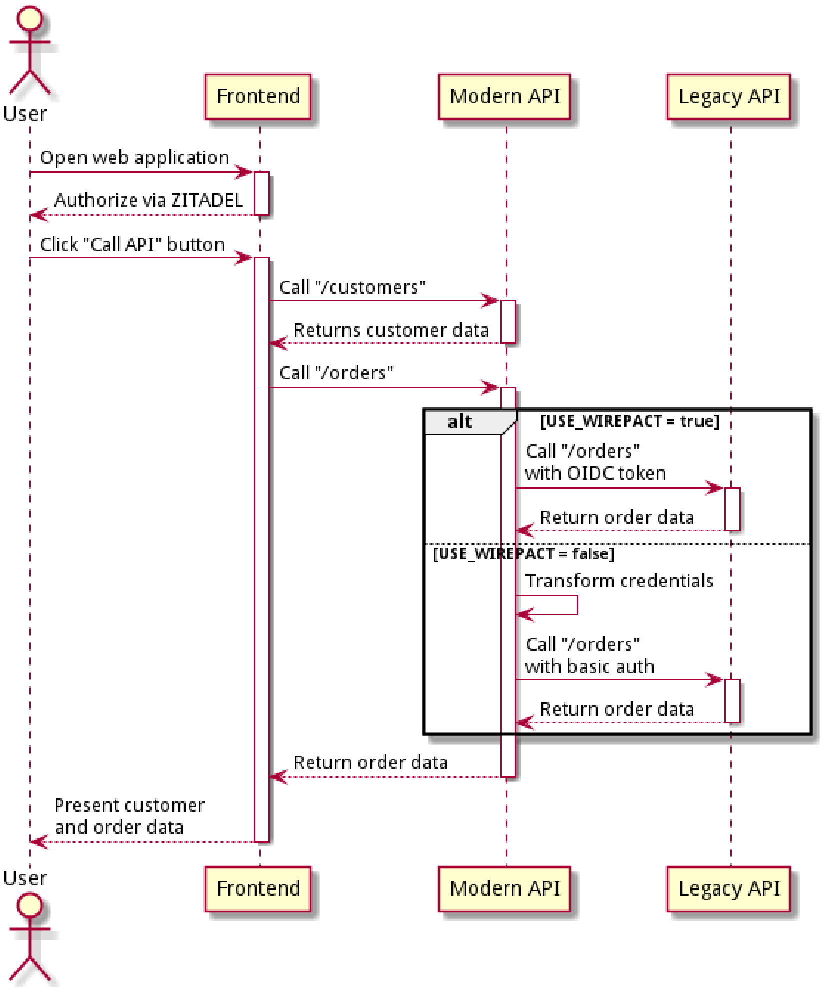

Figure 23: Sequence Diagram of the Communication in the Case Study

In Figure 23, we show the process of a user call in the demo application. The user opens the web application and authenticates himself with ZITADEL. After that, the user is presented with the application and can click the "Call API" button. The frontend application calls the modern backend API with the access token from ZITADEL and asks for customer and order data. The customer data is present on the modern API, therefore it is directly returned. To fetch the order data, the modern service relies on a legacy application which is only capable of Basic Authentication.

Depending on the configuration (i.e. the environment variable `USE_WIREPACT`), the modern service will call the legacy application with either transformed basic authentication credentials (when `USE_WIREPACT=false`) or with the ZITADEL access token (`USE_WIREPACT=true`). Either way, the legacy API receives basic authentication credentials in the form of `<username>:<password>` and returns the data.

### 4.6.2 Automation Engine for Applications

As explained in Section 4.4.2, the automation engine is generally optional. If omitted, the user is responsible for configuring the proxy and the translator. In the PoC, the automation engine is a Kubernetes Operator written with the .NET SDK in C#. The source of the PoC Operator resides on GitHub: https://github.com/WirePact/poc-operator. The Operator (automated and customized management of resources in Kubernetes, see Section 2.3) intercepts events for `Deployments` and `Services`. To update services and deployments in the PoC, an annotation (key-value storage in the metadata of an object in Kubernetes) is used. In future work, the Operator may react to Custom Resource Definitions (CRD) as well.

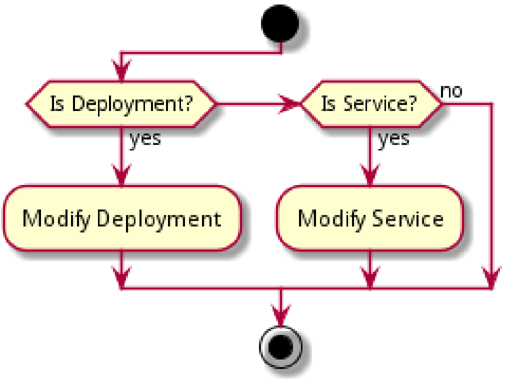

Figure 24: Activity Model for Kubernetes Resources in the Automation Engine

Figure 24 gives an overview of the process that an event of the Kubernetes API completes. When the Operator receives a notification by Kubernetes that a service or a deployment was created or modified, the Operator determines the type and uses the specific controller to reconcile the resource. If the entity is a deployment/service and is relevant for the authentication mesh, the Operator will modify the deployment/service.

In the case of a deployment, the first step of the Operator is to determine if the entity is relevant for the authentication mesh with the concept in Figure 12. If the deployment contains the annotation `ch.wirepact/port` in its metadata, it is automatically part of the mesh. If the deployment is already configured, further reconfiguration is skipped. Otherwise, the Operator fetches the already configured ports of the deployment, and generates two additional ports. One port is used for the Envoy sidecar while the other is configured for the translator sidecar. The next step is to generate and store the Envoy configuration in a Kubernetes `ConfigMap`. Last, the sidecars are injected into the deployment configuration and the Kubernetes client stores the modified manifest.

When reconciling a service, the service counts as relevant if the annotation `ch.wirepact/deployment` is present in the metadata of the service. The value of this annotation stores the deployment object to which the service should point. The Operator reads the annotations on the service to determine the port in question and searches for the port in its manifest. The port will receive a new “target port” that points to the Envoy port of the deployment. Last, the Kubernetes client will store the changed service.

### 4.6.3 Network and Routing Proxy for Communication

In the PoC, the proxy sidecar is an Envoy proxy with its configuration injected by the automation engine. The Operator injects the sidecar whenever a `Deployment` is created or updated via the Kubernetes API. A `ConfigMap` with the envoy configuration is created during reconciliation.

Two parts of the envoy configuration are crucial. First, the `filter_chain` of the inbound traffic listener contains a list of `http_filters`. Within this list of filters, the external authorization filter is added to force Envoy to check if a request is allowed or not:

```
# ... more config
http_filters:
  - name: envoy.filters.http.ext_authz
    typed_config:
      '@type': type.googleapis.com/
        envoy.extensions.filters.http.
        ext_authz.v3.ExtAuthz
      transport_api_version: v3
      grpc_service:
        envoy_grpc:
          cluster_name: auth_translator
        timeout: 1s
      include_peer_certificate: true
  - name: envoy.filters.http.router
# ... more config
```

Second, via the configured name (`auth_translator`), the external authorization service must be added to the `clusters` list:

```
# ... more config
- name: auth_translator
  connect_timeout: 0.25s
  type: STATIC
  typed_extension_protocol_options:
    envoy.extensions.upstreams.http.v3.HttpProtocolOptions:
      '@type': type.googleapis.com/
        envoy.extensions.upstreams.http.
        v3.HttpProtocolOptions
      explicit_http_config:
        http2_protocol_options: {}
  load_assignment:
    cluster_name: auth_translator
    endpoints:
      - lb_endpoints:
          - endpoint:
              address:
                socket_address:
                  address: 127.0.0.1
                  port_value: <<PORT_VALUE>>
# ... more config
```

This configures Envoy to find the external authorization service on the local loopback IP on the configured port. Since the transformer uses gRPC (`grpc_service: envoy_grpc: ...` in the filter config), http2 must be enabled for the communication. In a productive environment, timeouts should be set accordingly.

### 4.6.4 Translator

The translator is the part of the PoC that performs the modification of HTTP headers per request. Since the intermediate DSL is not implemented in the PoC, the translator converts an access token to static basic authentication credentials. If any error occurs or the translator call exceeds ten seconds, Envoy returns a HTTP 403 Forbidden message by default. The source code is on GitHub: https://github.com/WirePact/poc-demo-translator.

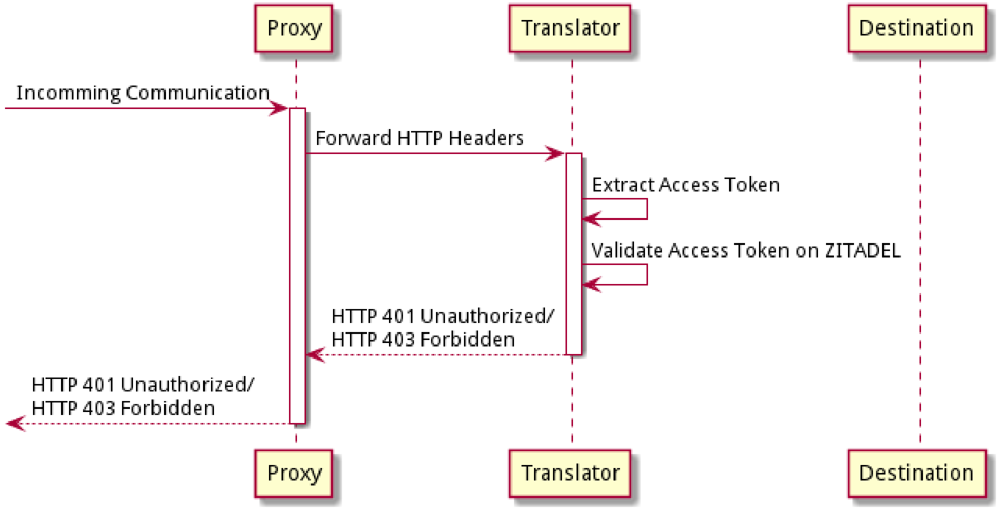

Figure 25: Communication with an Invalid Access Token

Figure 25 shows the sequence for an access token that is not valid. Envoy forwards the HTTP headers to the translator that extracts the `Authorization` header. If it is not a `Bearer` access token, or if the validation with ZITADEL fails (if the token is invalid or has expired), the translator returns an `Unauthorized` (HTTP 401) or `Forbidden` (HTTP 403) response depending on the status. The `Unauthorized` status is returned when no access token is provided (i.e. the HTTP header is missing). `Forbidden` is used if the token is invalid. In either case, Envoy will return the returned status code to the caller and terminates the request. The destination application does not receive any communication or notification about this event.

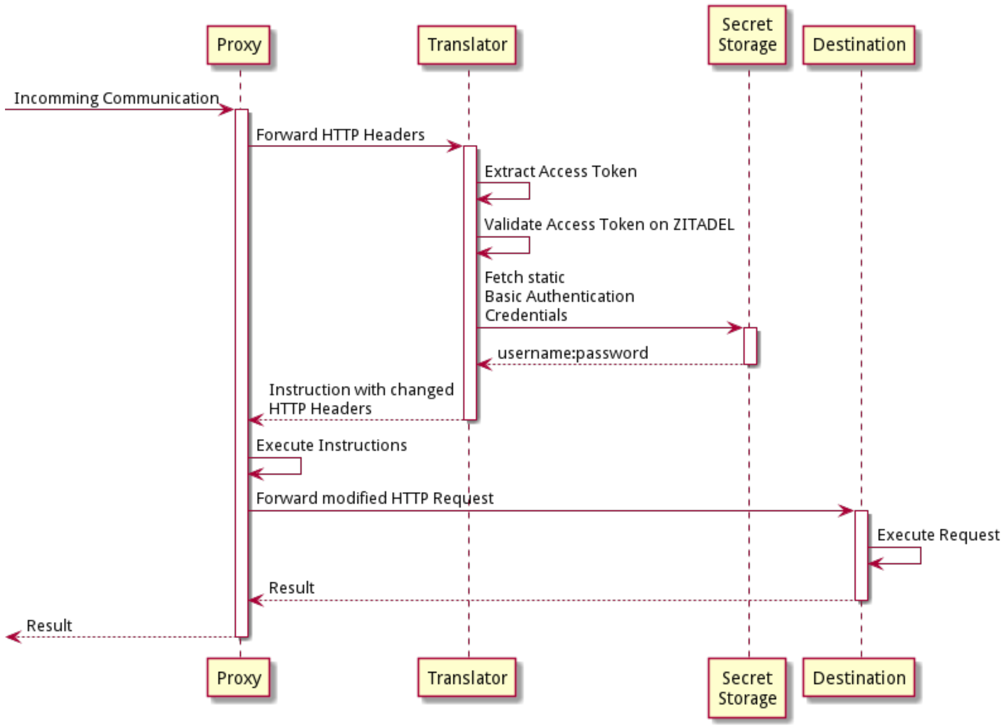

Figure 26: Communication with a Valid Access Token

In contrast to Figure 25, the sequence in Figure 26 shows the success path of a communication. If the given access token is valid, the translator fetches the static Basic Authentication credentials (i.e. username and password) from the secret storage. The secret storage in the PoC is a simple Kubernetes Secret. The received credentials are then transformed in the correct encoded Basic Authentication format (as described in RFC7617). The translator returns an instruction set for Envoy to process the HTTP request. Envoy executes the instructions and forwards the call to the destination and returns the response — if any.

When the translator decides that the request is unauthorized or forbidden, it returns a `DeniedResponse` to Envoy. The response is encoded in a binary "Protocol Buffers"[20].

In contrast to the rejected response, an accepting response may include modifications for HTTP headers. It is possible to add new, modify, and remove headers from the request for the upstream (i.e. the destination of the request), as well as adding additional or modifying headers for the downstream (i.e. the source of the request when the result is returned).

[20] Binary Data Format by Google: https://developers.google.com/protocol-buffers

# 5 Evaluation

This section evaluates the concepts and the architecture of Section 4. The main goal is to show that the proposed solution can improve the current situation and does not introduce security issues when used.

## 5.1 Architecture against Requirements

To show that the architecture of the distributed authentication mesh has the potential to improve the developer experience and the current situation with legacy or third-party software, we compare the architecture against the non-functional requirements established in Section 4.

### 5.1.1 NFR 1: Improve Security

> NFR1: First and foremost, the solution must not be less secure than current solutions.

Without the distributed authentication mesh, credentials like access tokens or basic authentication credentials are transmitted in the HTTP headers. This is a well-known way of authorizing requests [18]. If the current standard is regarded “secure” — not judging by the authorization scheme — then the mesh is secure as well. It even improves security by hiding the original credentials.

In the PoC, the credentials are still transmitted. The PoC is responsible to show that modifying HTTP headers during a request is possible. Securing the implementation of the concept is not part of this project.

### 5.1.2 NFR 2: Secure implementation

> NFR 2: The solution must adhere to current best practices and security mechanisms. Furthermore, it must be implemented according to the standards of the practice to mitigate security issues as stated in the OWASP Top Ten (https://owasp.org/www-project-top-ten).

The following list shows the OWASP top ten security issues with the comparison to the architecture:

1. Injection: The distributed authentication mesh does not use any database or LDAP features. Thus, there is no attack vector for an injection attack.

2. Broken Authentication[21]: The mesh does not implement a security scheme by itself. The only part that can be targeted by broken authentication attacks is the transformer. Developers of each translator are responsible to adhere to the OWASP principles and state-of-the-art security mechanisms.
3. Sensitive Data Exposure: The transmitted user identity must not include sensitive data. Sensitive data, such as financial or healthcare data is not part of a user identity and not needed for the mesh. Data, such as the user id or the name of a user, may be transmitted.
4. XML External Entities: The mesh does not use XML.
5. Broken Access Control: The mesh only provides valid credentials for the target system. It is not responsible for the authorization and enforcement of rules. The application that uses the mesh is responsible to enforce authorization rules.
6. Security Misconfiguration: The mesh does not directly influence used authentication schemes. However, the translators are directly responsible to use the correct HTTP headers for the target authentication mechanism. Developers of translators therefore responsible to correctly implement the authentication schemes.
7. Cross-Site Scripting (XSS): The mesh is not part of any public-facing application. No code gets executed. Therefore, XSS is not possible.
8. Insecure Deserialization: Under the assumption that JWT is used to transmit the user identity between participating elements, this flaw is negated. The validation and deserialization of JSON does not execute any code since JSON cannot transmit executable data. The translator must not execute any data it receives from the JWT.
9. Using Components with Known Vulnerabilities: This may be an issue if developers of translators do not update their software. The translator is the "moving part" of the mesh, which can be implemented by other developers as well. Developers must update their translators to eliminate this issue.
10. Insufficient Logging & Monitoring: This issue cannot be validated based on the architecture of the mesh. Since there is no production-ready implementation, logging is a part of the future work.

As the list above shows, the architecture does eliminate or out-source the OWASP issues. Translators must be implemented with special care to adhere to the security standards.

### 5.1.3 NFR 3: Generic Usage

> NFR 3: The concept of the solution is applicable to cluster orchestration software other than Kubernetes. The architecture provides a general way of solving the stated problem instead of giving a proprietary solution for one vendor. The concept should even be realizable for non-orchestration platforms like a Windows operating system.

[21] Broken Authentication relates to incorrectly implemented authentication and session management. This would allow attackers to compromise sessions and passwords.

The abstract architecture in Section 4 is generic. All components may be implemented on any platform and with any programming language of choice. The automation engine is optional so that the proposed concept may be implemented as a macOS or Windows software. There is no special requirement for any part of the mesh that ties the solution to a specific vendor.

### 5.1.4 NFR 4: Performance Impact

> NFR 4: The translation of the credentials should not extensively impact the timeframe of an arbitrary request. In production mode, the additional time to check and transform the credentials should not exceed 100ms. This is a general recommendation and some authentication mechanism may exceed the stated 100ms.

The architecture does not give hints about the effective performance impact. This generally depends on the used authentication scheme and the implementation of the transformer. Each transformer is responsible to achieve this goal. The solution is — theoretically — not limited in execution time, but to function as a production-ready solution, it must not impact the execution time of requests significantly.

### 5.1.5 NFR 5: Modularity

> NFR 5: The solution must be extensible with additional “translators” that provide the means of transforming the given credentials to other target formats.

The architecture shows that the translator is a component that is orchestrated by the automation engine. The translators should target one specific authentication scheme and can be implemented in any language or framework. They must only adhere to the principles of the mesh. It is not defined how the communication between the proxy and the translator takes place. In the PoC, Envoy (as the proxy) has a well-defined gRPC definition for such communication. Further work may contain the definition of translators for the automation engine. Using Envoy and the gRPC definition is a feasible option to implement a production-ready version of the mesh when using a cloud environment.

### 5.1.6 NFR 6: Integration into Infrastructure

> NFR 6: The solution may run with or without a service mesh. It is a goal that the solution can run without a service mesh to reduce the overall complexity, but if a service mesh is already in place, the solution must be able to work with the provided infrastructure.

The shown architecture in Section 4 does not interfere with a service mesh. If a service mesh is already deployed on a cloud environment, the automation engine must configure/reuse the parts that are already given by the service mesh.

### 5.1.7 NFR 7: Scalability

> NFR 7: The architecture must be scalable. In a cloud-native environment, the application that is enhanced may be scaled. Therefore, the solution must be able to scale with the application as well.

Section 4 shows that the automation engine does enhance Kubernetes pods. A pod is one unit of deployment. When a pod is scaled by Kubernetes, all containers in the pod do scale as well. Since all parts of the mesh are complete packages, they do scale with the pod.

### 5.1.8 NFR 8: Separation of Concerns

> NFR 8: Each translator should only handle one authentication scheme to ensure separation of concerns and scalability of the whole solution.

The architecture does not define the effective implementation of the translators. Each translator can be written in any language or framework. The responsibility to adhere to the separation of concerns is handed over to the developers of translators.

### 5.1.9 NFR 9: No Data-Transfer

> NFR 9: The solution depends on an external software for data transmission. The solution must not interfere with the data plane. Error handling of the data plane is handled by the external application.

The proxy and translator only modify HTTP headers. The effective transmission of the data between the parties is not part of the authentication mesh. As such, error handling for the transmission is also out-sourced to the used proxy software.

### 5.1.10 NFR 10: Error Handling

> NFR 10: The solution handles errors in the translation and the automation engine according to the architectural description.

All parts of the distributed authentication mesh rely on external software, except for translators. The automation engine is optional and if it fails, the underlying system is responsible to restart the engine. The source and destination services are not in the responsibility of the mesh by themselves. In the PoC, the proxy is an Envoy instance that contains error handling for the data-transfer. In addition, Kubernetes provides error handling for non-running applications and is responsible for the running state of the applications.

The only critical elements in the authentication mesh are translators. Since they are custom implementations, they must contain error handling for the requests. Translators receive HTTP headers and must parse some user identity out of it. If a translator is not able to construct the necessary HTTP headers for the destination, the request must fail. If any other error occurs (e.g. user repository not accessible) the request must fail as well.

Section 4 states that in the case of a timeout, error, or invalid data, the request must be blocked by the translator. Only valid requests must be let through to the destination.

## 5.2 Leaking Credentials and Developer Experience

As stated in Section 3, when applications with diverging authentication schemes communicate with each other, they must transmit credentials to the destination. Otherwise, it would not be possible to authenticate a user in each system.

The distributed authentication mesh replaces the need of effective credentials in communication with federated identity. Similar to SAML (explained in Section 4), an encoded identity is transmitted with the request instead of user credentials such as passwords or access tokens. This identity is then translated to effective credentials in the translator and ultimately forwarded to the target application.

Another identified problem in Section 3 is the introduction of code changes when the use case for the authentication mesh arises. To enable “modern software” to talk with “legacy software” (or third-party software), most likely the modern software will implement the translation logic. This may introduce bugs and does not scale when the service landscape grows.

The proposed concept enables developers to declaratively (via configuration) transform such credentials between applications. When used in a cloud environment, the automation engine can take care of all moving parts. With the solution in place, a developer is only required to configure the application as part of the mesh and the automation engine will inject the needed proxy and translators. After the automation step has taken place, the application is enhanced with additional authentication schemes without implementing the effective translation.

Therefore, the distributed authentication mesh enhances the general security of a system by removing the need of transmitting credentials to other services. Also, the developer experience is improved by allowing software developers to configuratively add authentication schemes to their software instead of manually developing conversion mechanisms for credentials.

# 6 Conclusions and Outlook

This report developed a potential solution to the problem of dynamic credential transformation in systems with diverging authentication mechanisms. In Section 1, a brief overview stated the problem and described the goal of the project.

Section 2 defined the scope of the project and explained various technologies and terms like "Kubernetes," "Operator Pattern," and "Sidecar Pattern." Additionally, Section 2 introduced vital information about authentication, authorization, and security standards required for the general understanding of this report.

Section 3 gave an overview of the current state of the art and the current problems. The maintainability of implementing multiple authentication schemes and the leakage of user credentials (like access tokens) onto the wire were identified as core problems.

To solve the stated problems in Section 3, a conceptional architecture was proposed in Section 4. To show the architecture within a practical environment, Section 4 also gave a platform-specific example of the architecture in Kubernetes. The concept of the distributed authentication mesh introduced a solution to the issues of diverging authentication schemes and leakage of credentials. When using a proxy component to intercept traffic from and to a service, a "translator" component can modify the HTTP headers of the requests. This removed the requirement of transmitting sensitive credentials over the wire, which fixed the problem of leaking credentials. The translator transforms outgoing credentials (for example an access token) to a common format. On the receiving side, the proxy intercepts the request and the translator converts the common format into the authentication scheme fitting the destination service. With the same procedure, the issue of implementing multiple authentication schemes in an application was addressed as well. It is possible to have multiple transformers and therefore serve a multitude of authentication mechanisms without introducing code changes to the applications.

The design of the distributed authentication mesh came close to the concept of SAML (Security Assertion Markup Language). While SAML provides a federated identity, it requires the participating services to implement the SAML protocol as well, the authentication mesh removes this requirement. The shown concept improves the developer experience by allowing dynamic credential transformation.

To validate if the concept is feasible, the Proof of Concept in Section 4 has shown that it is possible to modify HTTP headers in-flight and therefore the core concept of the architecture is generally possible. Furthermore, Section 5 checked if the given requirements and goals/non-goals were achieved with the proposed architecture. The evaluation shows that the solution is able to enhance general security by preventing the leakage of credentials.

As complementation to the main delivery of this project — the concept of the distributed authentication mesh — teaching material was created that can be found in Appendix C. It targets the topic of "Kubernetes Operators and how to create them." This material may be used to introduce people to the operator pattern and helps to create a custom operator with an SDK.

The goal of the future work is to provide a federated authentication with secured communication without leakage of credentials out of the trust zone. The analysis, definition, and implementation of a common identity format for the transmission in future work complements the concept of the distributed authentication mesh. The concepts of the mesh will be used to implement a production-ready version of the authentication mesh in Kubernetes.

With the implementation of the authentication mesh in Kubernetes, various use cases can be covered. As an example, in the finance sector, banking APIs tend to use varying authentication schemes and do not wish to change their applications. The authentication mesh improves this situation by covering the dynamic transformation of credentials to the respective format. The concept of the distributed authentication mesh can be implemented as an application that runs directly on an operating system to provide a federated identity into a company network. It negates the requirement of implementing technologies like SAML in each application that is accessed in this company network.

When regarding the current trends, applications will become more heterogeneous in the future. Authentication protocols come and go, and it is not likely that one particular standard will solve all issues. The concepts of this project contribute to sustainability and reusability in the security world.

# Bibliography

[1] CNCF, “Production-grade container orchestration,” *Kubernetes Website.* https://github.com/kubernetes/website; GitHub, Mar. 2021.Available: https://kubernetes.io/

[2] Creative Commons, “Attribution 4.0 international (CC BY 4.0).” https://creativecommons.org/licenses/by/4.0/, 2021.

[3] J. Dobies and J. Wood, *Kubernetes operators: Automating the container orchestration platform.* O’Reilly Media, Inc., 2020.

[4] B. Burns and D. Oppenheimer, “Design patterns for container-based distributed systems,” Jun. 2016.Available: https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/burns

[5] W. Li, Y. Lemieux, J. Gao, Z. Zhao, and Y. Han, “Service mesh: Challenges, state of the art, and future research opportunities,” in *2019 IEEE international conference on service-oriented system engineering (SOSE)*, 2019, pp. 122–1225. doi: 10.1109/SOSE.2019.00026.

[6] F. Montesi and J. Weber, “Circuit breakers, discovery, and API gateways in microservices,” *CoRR*, vol. abs/1609.05830, 2016,Available: http://arxiv.org/abs/1609.05830

[7] J. Reschke, “The ’Basic’ HTTP authentication scheme,” Internet Engineering Task Force IETF, RFC, 2015.Available: https://tools.ietf.org/html/rfc7617

[8] D. Hardt and others, “The OAuth 2.0 authorization framework,” Internet Engineering Task Force IETF, RFC, 2012.Available: https://tools.ietf.org/html/rfc6749

[9] N. Sakimura, J. Bradley, M. Jones, B. De Medeiros, and C. Mortimore, “Openid connect core 1.0,” The OpenID Foundation OIDF, Spec, 2014.Available: https://openid.net/specs/openid-connect-core-1_0.html

[10] S. Rose, O. Borchert, S. Mitchell, and S. Connelly, “Zero trust architecture,” National Institute of Standards; Technology, 2019.

[11] N. Kratzke and R. Peinl, “ClouNS - a cloud-native application reference model for enterprise architects,” in *2016 IEEE 20th international enterprise distributed object computing workshop (EDOCW)*, 2016, pp. 1–10. doi: 10.1109/EDOCW.2016.7584353.

[12] M. Feathers, *Working effectively with legacy code.* USA: Prentice Hall PTR, 2004.

[13] Istio Authors, “Mutual TLS migration,” *Istio.* Mar. 2021.Available: https://istio.io/latest/docs/tasks/security/authentication/mtls-migration/

[14] Istio Authors, “External authorization,” *Istio.* Mar. 2021.Available: https://istio.io/latest/docs/tasks/security/authorization/authz-custom/

[15] F5 Inc. Authors, “Authentication based on subrequest result,” *NGINX.* 2021.Available: https://docs.nginx.com/nginx/admin-guide/security-controls/configuring-subrequest-authentication/

[16] N. Naik and P. Jenkins, “Securing digital identities in the cloud by selecting an apposite federated identity management from SAML, OAuth and OpenID connect,” in *2017 11th international conference on research challenges in information science (RCIS)*, 2017, pp. 163–174. doi: 10.1109/RCIS.2017.7956534.

[17] P. Siriwardena, “Mutual authentication with TLS,” in *Advanced API security: Securing APIs with OAuth 2.0, OpenID connect, JWS, and JWE*, Berkeley, CA: Apress, 2014, pp. 47–58. doi: 10.1007/978-1-4302-6817-8_4.

[18] H. Nielsen, R. T. Fielding, and T. Berners-Lee, “Hypertext Transfer Protocol – HTTP/1.0,” RFC 1945; RFC Editor, RFC 1945, May 1996. doi: 10.17487/RFC1945.

[19] M. B. Jones, Bradley John, and N. Sakimura, “JSON web token (JWT),” Internet Engineering Task Force IETF, RFC, May 2015.Available: https://tools.ietf.org/html/rfc7519

[20] C. Bormann and P. E. Hoffman, “Concise Binary Object Representation (CBOR),” Internet Engineering Task Force IETF, RFC, 2020. doi: 10.17487/RFC8949.

[21] D. Cooper, S. Boeyen, S. Santesson, T. Polk, R. Housley, and S. Farrell, “Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile,” Internet Engineering Task Force IETF, RFC 5280, May 2008. doi: 10.17487/RFC5280.

[22] P. Bryan and M. Nottingham, “Javascript object notation (JSON) patch,” Internet Engineering Task Force IETF, RFC, Apr. 2013.Available: https://tools.ietf.org/html/rfc6902

[23] B. Selic, “Controlling the controllers: What software people can learn from control theory,” *IEEE Software*, vol. 37, no. 6, pp. 99–103, 2020, doi: 10.1109/MS.2020.3006970.

# Appendix A: Common Kubernetes Terminology

In Table A.1, we state the most common Kubernetes terminology. The table provides a list of terms that is used to explain concepts like the Operator pattern.

Table A.1: Common Kubernetes Terminology

| Term | Description |
|---|---|
| Docker | Container runtime. Enables developers to create container images of applications. Those images are then run in an isolated environment. Docker images are often used in Kubernetes to define the application that Kubernetes should run. |
| Kustomize | “Kustomize” is a special templating CLI to declaratively bundle Kubernetes manifests. It consists of a `kustomization.yaml` and various referenced manifest `yaml` files. It is declarative and does not allow dynamic structures. It helps administrators template applications for Kubernetes. |
| Container | Smallest possible unit in a deployment. Contains the definition of the workload. A container consists of a container image, arguments, volumes and other specific information to carry out a task. |
| Pod | Composed of multiple containers. It is ran by Kubernetes as an instance of a deployment. Pods may be scaled according to definitions or “pod scalers.” Highly coupled tasks are deployed together in a pod (i.e. multiple coupled containers in a pod). |
| Deployment<br>Daemonset<br>Statefulset | A “Deployment” is a managed instance of a pod. Daemonsets and Statefulsets are variants of deployments. Kubernetes will run the described pod(s) with the desired replica count on the best possible worker node. Deployments may be scaled with auto-scaling mechanisms. |
| Service | A service enables communication with one or multiple pods. The service contains a selector that points to a certain number of pods and then ensures that the pods are accessible via a DNS name. The name is typically a combination of the service name and the namespace (e.g. `my-service.namespace`). |
| Ingress | Incoming communication and data flow into a component. Furthermore, an “Ingress” is a Kubernetes object that defines incoming communication and configures an API gateway to route traffic to specific services. |
| Egress | Outgoing communication. Egress means communication from a component to another (when the component is the source). |

| Term | Description |
|---|---|
| Resource | A resource is something that can be managed by Kubernetes. It defines an API endpoint on the master node and allows Kubernetes to store a collection of such API objects. Examples are: `Deployment`, `Service` and `Pod`, to name a few of the built-in resources. |
| CRD | A Custom Resource Definition (CRD) enables developers to extend the default Kubernetes API. With a CRD, it is possible to create own resources which create an API endpoint on the Kubernetes API. An example of such a CRD is the `Mapping` resource of Ambassador[22]. |
| Operator | An operator is a software that manages Kubernetes resources and their lifecycle. Operators may use CRDs to define custom objects on which they react when some event (`Added`, `Modified` or `Deleted`) triggers on a resource. For a more in-depth description, see Section 2.3. |
| Watcher | A watcher is a constant connection from a client to the Kubernetes API. The watcher defines some search and filter parameters and receives events for found resources. |
| Validator | A validator is a service that may reject the creation, modification or deletion of resources. |
| Mutator | Mutators are called before Kubernetes validates and stores a resource. Mutators may return JSON patches **RFC6902** [22] to instruct Kubernetes to modify a resource prior to validating and storing them. |

[22] https://www.getambassador.io/

# Appendix B: Installation of the PoC

This section shows how to install the case study locally. The installation guide is also hosted on GitHub (https://github.com/WirePact/wirepact-poc). The installation consists of the operator and the case study with three application parts. To access the application, Ambassador acts as API gateway.

To begin the installation of the PoC, a Kubernetes environment is needed. On Windows and Apple devices, Docker Desktop with Kubernetes[23] is recommended. Other environments, for example minikube[24], work as well. The next step is to install Ambassador as API gateway with the shell script `./Kubernetes/case-study/install-ambassador.sh`. On Windows, the Subsystem for Linux or the git bash can be used to execute the shell script. Otherwise, the PowerShell can be used to execute the `kubectl` commands in the shell script one by one.

For the last step, the `Kustomize`[25] executable is required. Change into the `Kubernetes` directory and run `kustomize build` to see the output of the `kustomization.yaml` file or `kustomize build | kubectl apply -f -` to build and directly apply the result to Kubernetes. This installs the operator and the case study. When everything is set up, the frontend application can be accessed via `https://localhost`, `https://kubernetes.docker.internal`, or `https://kubernetes.local` depending on the host's config of the machine.

To be able to log in into the frontend application, any ZITADEL account may be used. It does not matter if the account is bound to an organization or resides in the global organization.

[23] https://docs.docker.com/desktop/kubernetes/

[24] https://minikube.sigs.k8s.io/docs/start/

[25] https://kustomize.io/

# Appendix C: Teaching Material for Kubernetes Operators

## Motivation

There is a variety of Kubernetes Operators. For example, the Prometheus Operator[26] which manages instances of Prometheus[27] in Kubernetes. A non-exhaustive list of Operators can be found on https://operatorhub.io. An Operator is not required to perform only one task.

Since Operators are an elegant tool to extend the capabilities of Kubernetes, developers may want to know how to create a custom Operator. This material gives an overview of the Operator pattern and a description of how Operators work. As an exercise, a custom Operator must be written with the help of an SDK. The solution to the custom Operator is implemented with C# and the .NET Operator SDK "KubeOps"[28].

## Learning Objectives

The Operator pattern[29] extends Kubernetes in a specified way [3]. One can extend the API of Kubernetes with custom resources and react to events of the resources. To be able to implement a custom Operator, the building blocks and concepts of the internal elements of an Operator must be known. An SDK helps to create an Operator, but when the Operator gets more complex, it may be vital to know how Operators work. Therefore, this material shows how an Operator works and how one can be built.

To summarize the learning objectives:

- One can explain the operator pattern and their parts with own words
- One can compare the pattern with alternative solutions
- One can build a custom operator with an SDK

## Kubernetes Operators and their Use

### What is an Operator?

An Operator is a piece of software that is designed to automate management of other software. It typically manages the lifecycle of another application [3]. As an example, the above-mentioned "Prometheus Operator" manages the lifecycle of "Prometheus." Normally, to fulfil their duty, Operators extend the API of Kubernetes by adding custom resource definitions.

[26] https://github.com/prometheus-operator/prometheus-operator

[27] https://prometheus.io/

[28] https://github.com/buehler/dotnet-operator-sdk

[29] https://kubernetes.io/docs/concepts/extend-kubernetes/operator/

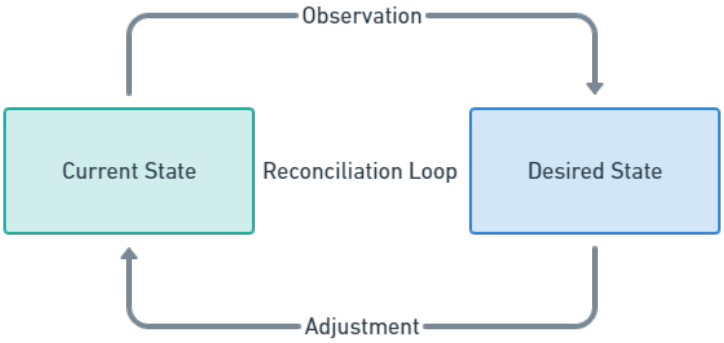

Figure C.1: The Kubernetes reconciliation loop. It can be compared to a basic feedback controll system that reacts based on system output with a controller [23].

To describe the "reconciliation loop" of an Operator, we consider Figure C.1. The reconciliation loop is the constant observation of the current state. When the current state diverges from the desired state, adjustments must be made to achieve the current state again. This loop is used by the Kubernetes API itself. As an example, if a user creates a deployment in Kubernetes, the API stores the deployment as the new desired state. The reconciliation loop checks if the deployment exists, and if it does not, it creates the deployment to reach the desired state.

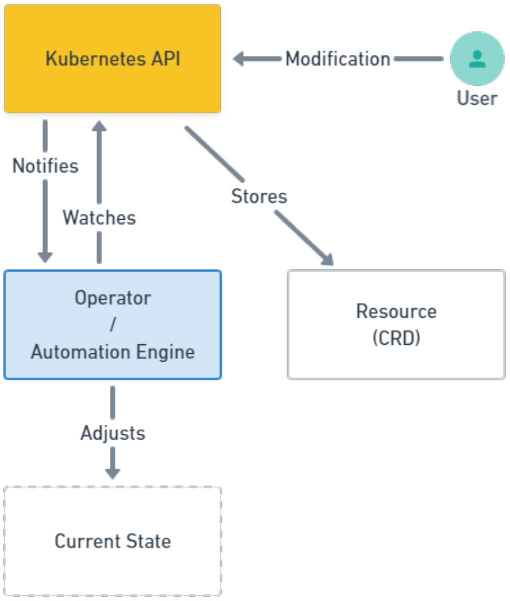

Figure C.2: Kubernetes Operator Pattern

The Operator pattern uses the reconciliation loop to manage a custom application or a custom use case. The pattern is shown in Figure C.2. When a user modifies resources (be it custom resources or predefined ones), the API stores the resources as the new desired state. The Operator gets notified by the API and can adjust elements in Kubernetes.

The Operator pattern can be used to manage entire applications, for example Prometheus or PostgreSQL database servers. Another use case of an Operator could include injecting logging collectors into each deployment in the cloud environment.

### How do Operators work?

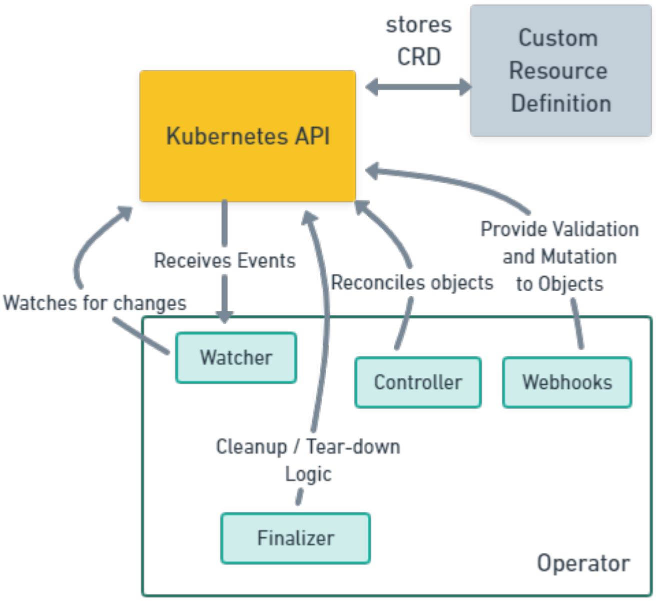

Figure C.3: Parts of an Operator and their Interaction with Kubernetes

Considering Figure C.3, the following objects exist in or around an Operator:

- **Watcher**[30]: The Operator registers one or multiple watchers with the Kubernetes API. This enables the Operator to receive events when a watched resource gets modified. A watcher can be namespaced or global and can watch one type of resource (e.g. Deployments, Services, or a custom resource).
- **Event**[31]: Events are the notification of the Kubernetes API that a watcher receives. Three relevant types of events exist:

[30] https://kubernetes.io/docs/reference/using-api/api-concepts/#efficient-detection-of-changes
[31] https://kubernetes.io/docs/reference/using-api/api-concepts/#efficient-detection-of-changes

  - *Added*: When a resource gets added to the watcher. This event is fired for each resource of the watched type when the watcher is registered.
  - *Modified*: When a resource that is already being watched gets modified.
  - *Deleted*: When a watched resource is removed from Kubernetes and the watcher.

- **Custom Resource Definition**[32]: A CRD defines non-standard objects that extend the API of Kubernetes. There exist resource definitions for all standard resources like deployments and services. A CRD enables developers to create custom resources which can be reconciled by an Operator.
- **Controller**[33]: Controllers are elements in an Operator that reconcile a specific CRD/resource. An Operator can contain multiple controllers and therefore manage multiple CRDs. A controller typically contains application logic to react to the events of the Kubernetes API.
- **Finalizer**[34]: A finalizer is a part of an Operator that enables asynchronous deletion processes in Kubernetes. When a resource contains finalizers in its metadata, the API will mark the resource as in pending deletion. An Operator may react to this state and can perform additional tasks, such as deleting a database or external resources. The Operator must then remove its finalizer entry. When all finalizers are removed, the resource is deleted. Otherwise, it will remain in the pending deletion state.
- **Mutation Webhook**[35]: A mutator (or mutation webhook) is an HTTP endpoint of an Operator. The endpoint will be called whenever a watched resource type is created/updated/deleted. A mutator may return an empty response to acknowledge the creation/modification/deletion of the resource, or it can patch the resource before the effective action is executed. The patch must be in the form of a JSON Patch, as defined in **RFC6902** [22]. As an example, one could create a profanity filter and remove "bad" usernames from resources. Mutators are **called in series** by the Kubernetes API.
- **Validation Webhook**[36]: In contrast to a mutation webhook, a validator (or validation webhook) may only accept or reject a resource. If multiple validators are registered for a certain type, they will be **called in parallel** by the Kubernetes API.

[32] https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/

[33] https://kubernetes.io/docs/concepts/architecture/controller/

[34] https://kubernetes.io/blog/2021/05/14/using-finalizers-to-control-deletion/

[35] https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/

[36] https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/

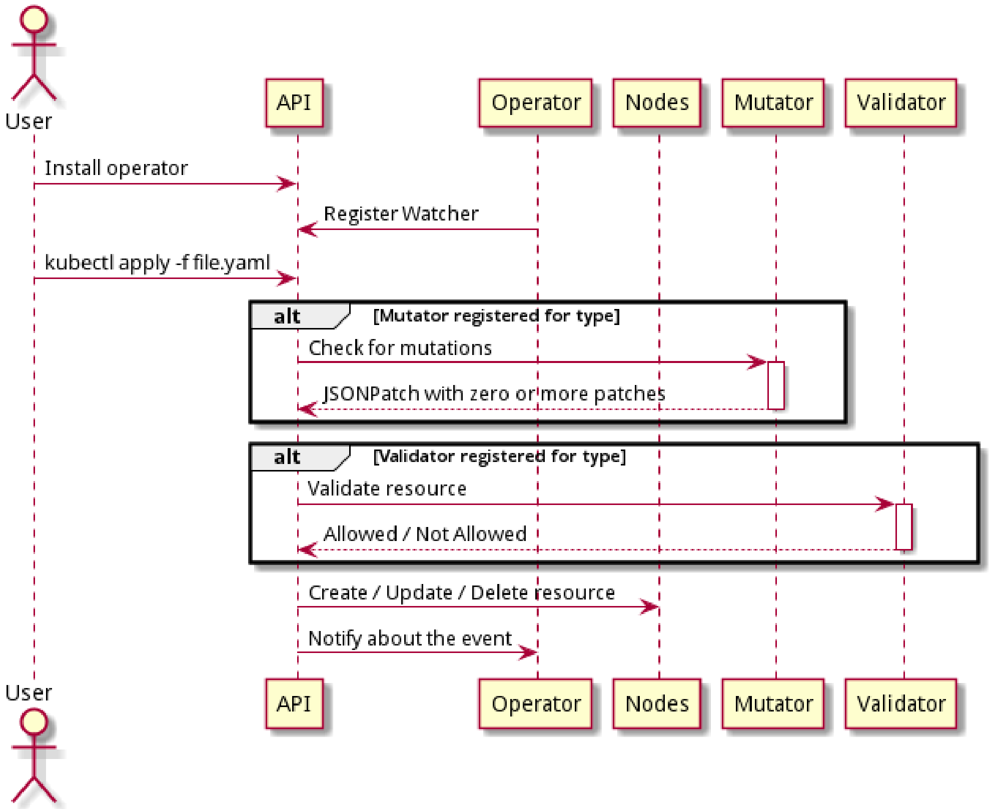

Figure C.4: Kubernetes Operator Workflow

To specify the general workflow in Figure C.2 in more detail, Figure C.4 depicts the concrete sequence of a reconciliation loop. An important note on admission webhooks (mutators/validators): if the Operator does not respond within ten seconds, the API will abort the creation/modification/deletion of the resource. This could lead to deadlocks when the Operator crashes or is not able to respond to the webhooks.

### What is an Operator SDK?

To help developers create custom Operators, SDKs provide the abstractions to perform the Kubernetes specific tasks. Depending on the SDK, several technical elements are abstracted like registering and error handling of the watchers. A non-exhaustive list of SDKs includes:

- KUDO[37]: A declarative Operator SDK that creates Operators based on declarative descriptions.

[37] https://kudo.dev/

- kubebuilder[38]: A GoLang based SDK that allows creating controllers and validation webhooks.
- OPERATOR SDK[39]: A multi-language SDK that allows GoLang based Operators, Helm based Operators, or Ansible based Operators by using hooks to execute scripts.
- Shell-operator[40]: Event driven script runner for Kubernetes. Operator SDK for shell scripts.
- Kopf[41]: "Kubernetes Operator Framework (Kopf)," is a Python based SDK with an immense feature set.
- KubeOps[42]: A .NET Operator SDK based on the principles of ASP.NET applications. Operators can be created with C# or F#.

## Exercise: Create a Custom Operator with an SDK

### TL;DR

1. Create an empty Operator with the KubeOps SDK and run it.
2. Create a CRD for a "WeatherLocation" and for "WeatherData."
3. Create the controllers for the CRDs and run/deploy the Operator. The use case is: The user can create a WeatherLocation object and the Operator should then fetch any weather data API and create WeatherData objects for each hour. When a specific amount of WeatherData elements are created, old objects must be deleted.

The examples and solutions are created with KubeOps in C#. When code is shown, the required "usings" are omitted. A possible solution can be found on GitHub: https://github.com/buehler/kubernetes-operator-exercise.

### Create and Run an empty Operator

Select an SDK and create an empty Operator and run it against a Kubernetes environment. One can use any Kubernetes environments, but it is advised to use a local instance like Docker Desktop with Kubernetes or minikube.

**Solution** KubeOps provides templates to create an Operator.

1. Install the templates: `dotnet new -i KubeOps.Templates::*`

[38] https://book.kubebuilder.io/
[39] https://operatorframework.io/
[40] https://github.com/flant/shell-operator
[41] https://kopf.readthedocs.io/en/stable/
[42] https://buehler.github.io/dotnet-operator-sdk/

2. Create the empty Operator: `dotnet new operator-empty -n WeatherOperator -o ./weather-operator`
3. Run the empty Operator against the Kubernetes environment

You should see the following log output:

```
info: KubeOps.Operator.Leadership.LeaderElector[0]
      Startup Leader Elector for operator "weatheroperator".
info: KubeOps.Operator.Leadership.LeaderElector[0]
      There was no lease for operator "weatheroperator".
      Creating one and electing "xxx" as leader.
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
```

The empty Operator in C# only registers the Operator logic with ASP.NET and starts the web server. The minimal config required to run consists of:

`Program.cs`:

```
static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });

await CreateHostBuilder(args).Build().RunOperatorAsync(args);
```

`Startup.cs`:

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddKubernetesOperator();
    }

    public void Configure(IApplicationBuilder app)
    {
        app.UseKubernetesOperator();
    }
}
```

### Create the Custom Resource Definition

Create the required objects in the Operator and create the CRD for Kubernetes. The CRD is the element that gets installed in the API of Kubernetes. The following two objects must be created:

- **WeatherLocation**: Object that contains the required data to query a weather API for weather data. As an example, if the OpenWeather API is used, the object should include latitude and longitude to identify the point of interest. Depending on the API you intend to use, you may need to add other fields.
- **WeatherData**: This object shall not be created by a user. It contains the "result" for a weather query. It must be linked to a WeatherLocation and should be cleaned up after 24 hours.

**Solution** `WeatherLocation`: To use the OpenWeather API, we only need the latitude and the longitude to create a weather call.

```
public class V1WeatherLocationSpec
{
    public double Latitude { get; set; }

    public double Longitude { get; set; }
}

public class V1WeatherLocationStatus
{
    public DateTime? LastCheck { get; set; }

    public string? Error { get; set; }
}

[KubernetesEntity(
    ApiVersion = "v1",
    Group = "kubernetes.dev",
    Kind = "WeatherLocation")]
[EntityScope(EntityScope.Cluster)]
public class V1WeatherLocation : CustomKubernetesEntity
    <V1WeatherLocationSpec, V1WeatherLocationStatus>
```

```
{
}
```

`WeatherData`: This object should be created by the Operator during runtime. It contains several elements of the weather call. The weather data object must be linked to a weather location.

```
public class V1WeatherDataSpec
{
    [AdditionalPrinterColumn]
    public string MainWeather { get; set; } = string.Empty;

    public string Description { get; set; } = string.Empty;

    [AdditionalPrinterColumn]
    public double Temperature { get; set; }

    public DateTime Sunrise { get; set; }

    public DateTime Sunset { get; set; }
}

[KubernetesEntity(
    ApiVersion = "v1",
    Group = "kubernetes.dev",
    Kind = "WeatherData")]
[EntityScope(EntityScope.Cluster)]
public class V1WeatherData : CustomKubernetesEntity<V1WeatherDataSpec>
{
}
```

KubeOps generates the CRDs found in the repository at https://github.com/buehler/kubernetes-operator-exercise/tree/main/config/crds.

These CRDs may now be installed into Kubernetes with `kubectl apply`.

### Reconcile the Custom Resource

As the Operator base and the CRDs are prepared, you are now to build the Operator logic. The Operator must fulfill the following requirements:

- A weather API call is executed each hour for a given weather location object
- The result of the API call is stored in Kubernetes as weather data object and linked to the weather location (owner reference)

- While reconciling, the Operator deletes old weather data objects (keep the last twelve)
- A validator checks if the longitude and latitude values are possible and denies the creation of the object if they are not within the boundaries
- A validator checks if a weather data object contains an owner reference

**Solution** Since it is not feasible to print the whole source code in this exercise, please find a possible solution on GitHub: https://github.com/buehler/kubernetes-operator-exercise.