Distributed Authentication Mesh*

Declarative Adhoc Conversion of Credentials

Christoph Bühler

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

*I would like to thank Dr. Olaf Zimmermann and Florian Forster. TODO.

TODO, this will contain the abstract of the project report.

Contents

1 Introduction **7**

2 Definitions and Boundaries **8**
2.1 Context . . . 8
2.2 Kubernetes . . . 9
2.2.1 Introduction . . . 9
2.2.2 Terminology . . . 10
2.2.3 Operator . . . 12
2.2.4 Sidecar . . . 14
2.2.5 Service Mesh . . . 14
2.3 Authentication and Authorization . . . 15
2.3.1 Basic (RFC7617) . . . 15
2.3.2 OpenID Connect (OIDC) . . . 16

3 State of the Art, the Practice and Deficiencies **19**
3.1 State of the Art . . . 19
3.2 The Practice . . . 22
3.3 Deficiencies . . . 22

4 Distributed Authentication Mesh **24**
4.1 Definition . . . 24
4.2 Requirements . . . 24
4.3 Non-Functional Requirements . . . 25
4.4 Contrast . . . 26
4.4.1 SAML . . . 26
4.4.2 WS-* . . . 27
4.5 Architecture . . . 28
4.5.1 Brief Description . . . 28
4.5.2 Use Case . . . 28
4.5.3 Solution Architecture . . . 29
4.5.4 Communication . . . 38
4.6 Implementation Proof of Concept (PoC) . . . 40
4.6.1 Showcase Application . . . 41
4.6.2 Operator . . . 44
4.6.3 Envoy Sidecar . . . 44
4.6.4 Translator . . . 45

5 Evaluation **46**

6 Conclusion **47**
6.1 Further Work 47

7 Bibliography **48**

Appendix A: Installation Guide for PoC **50**

Appendix B: Teaching Material for Kubernetes Operators **51**

List of Tables

2.1 Common Kubernetes Terminology 10

4.1 Functional Requirements 24
4.2 Non-Functional Requirements 25

List of Figures

2.1 Kubernetes Container Evolution 9
2.2 Kubernetes Operator Workflow 13
2.3 Example of a sidecar container 14
2.4 OIDC code flow 17

3.1 Microservice Architecture with legacy components 20
3.2 Current process of legacy communication 21

4.1 Solution Architecture 30
4.2 Automation Architecture 32
4.3 Automation Process 33
4.4 PKI Architecture 34
4.5 PKI Process 35
4.6 Networking Architecture 36
4.7 Inbound Networking Process 36
4.8 Outbound Networking Process 37
4.9 Translator Architecture 38
4.10 Translator Process 39
4.11 Component Diagram of the Showcase Application 42
4.12 Sequence Diagram of the Showcase Call 43

1 Introduction

Modern cloud environments solve many problems like the discovery of services and data transfer or communiation between services in general. One modern way of solving service discovery and communication is a Service Mesh, which introduces an additional infrastructure layer that manages the communication between services (Li et al. 2019, sec. 2).

However, a specific problem is not solved yet: "dynamic" trusted communication between services. When a service, that is capable of handling OpenID Connect (OIDC) credentials, wants to communicate with a service that only knows Basic Authentication that originating service must implement some sort of conversion or know static credentials to communicate with the basic auth service. Generally, this introduces changes to the software of services. In small applications which consist of one or two services, implementing this conversion may be a feasable option. If we look at an application which spans over a big landscape and a multitude of services, implementing each and every possible authentication mechanism and the according conversions will be error prone work and does not scale well[1].

The goal of the project "Distributed Authentication Mesh" is to provide a solution for this problem.

Prerequisits:

- Docker knowledge
- Microservice knowledge

  TODO.

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

2 Definitions and Boundaries

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 Context

TODO: restructure to have better reading flow

This project addresses the specific problem of declarative conversion of credentials to ensure authorized communication between services. The solution may be runnable on various platforms but will be implemented according to Kubernetes standards. Kubernetes[1] is an orchestration platform that works with containerized applications. The solution introduces an operator pattern, as explained in Section 2.2.3

The deliverables of this and further projects may aid applications or APIs to communicate with each other despite different authentication mechanisms. As an example, this could be used to enable a modern web application that uses OpenID Connect (OIDC) as the authentication and authorization mechanism to communicate with a legacy application that was deployed on the Kubernetes cluster but not yet rewritten. This transformation of credentials (from OIDC to Basic Auth) is done by the solution of the projects instead of manual work which may introduc code changes to either service.

This specific project provides a proof of concept (PoC) with an initial version on a GitHub repository. The PoC demonstrates that it is possible to instruct an Envoy[2] proxy to communicate with an injected service to modify authentication credentials in-flight.

To use the proposed solution of this project, no service mesh or other complex layer is needed. The solution runs without those additional parts on a Kubernetes cluster. To provide service discovery, the default internal DNS capabilities of Kubernetes are sufficient.

[1] https://kubernetes.io/
[2] https://www.envoyproxy.io/

2.2 Kubernetes

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 (CNCF 2021).

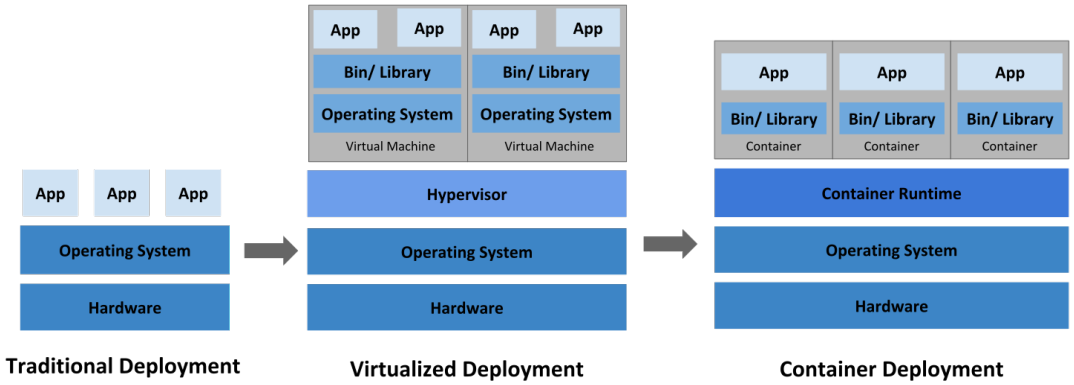

Figure 2.1: Container and Deployment Evolution. Description of the evolution of deployments as found on the documentation website of Kubernetes (CNCF 2021). This image is licensed under the CC BY 4.0 license (Creative Commons 2021).

According to Kubernetes, the way of deploying applications has evolved. As shown in Figure 2.1, the "Traditional Era" was the time, when applications were deployed via FTP access and started manually (e.g. on an Apache webserver). 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 (CNCF 2021).

Kubernetes is a major player, among others like "Docker Swarm" or "Cloud Foundry," in "Container Deployment" as seen in Figure 2.1 and supports teams with the following features according to the documentation (CNCF 2021):

- **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
- **Automatec 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 configurational data that may change the behaviour of a workload

The list of features is not complete. There are many concepts in Kubernetes which 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 to reach the desired state depicted in the applications manifest files. Examples of such manifests can be viewed at: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#creating-a-deployment.

2.2.2 Terminology

In Table 2.1, we state the most common Kubernetes terminology. The table provides a list of terms that will be used to explain concepts like the operator pattern in Section 2.2.3.

Table 2.1: Common Kubernetes Terminology

Term	Description
Docker	Container runtime. Enables developers to create 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 to 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.

Term	Description
Pod	Composed of multiple containers. 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	A deployment is a managed instance of a pod. Kubernetes will run the described pod with the desired replica count on the best possible worker node. Deployments may be scaled with auto-scaling mechanisms.
Service	A service enables communciation 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 accessable via a DNS name. The name is typically a combination of the servicename and the namespace (e.g. `my-service.namespace`).
Ingress	Incomming communication and data-flow into a component. Furthermore an "Ingress" is a Kubernetes object that defines incomming 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).
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 creates an API endpoint on the Kubernetes API. An example of such a CRD is the `Mapping` resource of Ambassador[3].
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 2.2.3.
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 the found resources.
Validator	A validator is a service that may reject the creation, modification or deletion of resources.

Term	Description
Mutator	Mutators are called before Kubernetes validates and stores a resource. Mutators may return JSON patches **RFC6902** (Bryan and Nottingham 2013) to instruct Kubernetes to modify a resource prior to validating and storing them.

TODO: UML of kubernetes parts

2.2.3 Operator

An operator in Kubernetes is an extension to the Kubernetes API itself. A custom operator typically manages the whole lifecycle of an appliction it manages (Dobies and Wood 2020). Such a custom operator can further be used to reconcile normal Kubernetes resources or any combination thereof.

Some examples of application operators are:

- Prometheus Operator[4]: Manages instances of Prometheus in a cluster
- Postgres Operator[5]: 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 operatorhub.io.

In Figure 2.2, we depict the general workflow of an event that is managed by an operator. When an operator is installed and running on a Kubernetes cluster, it registers "Resource Watchers" with the API and receives notifications when the master node modifies resources a watched resource. 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 in an operator.

When the user interacts with the Kubernetes API (for example 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 the notification about the watched resource and may interact with the event. Such an action may include to update resources, create more resources or even delete other instances.

[3]https://www.getambassador.io/
[4]https://github.com/prometheus-operator/prometheus-operator
[5]https://github.com/zalando/postgres-operator

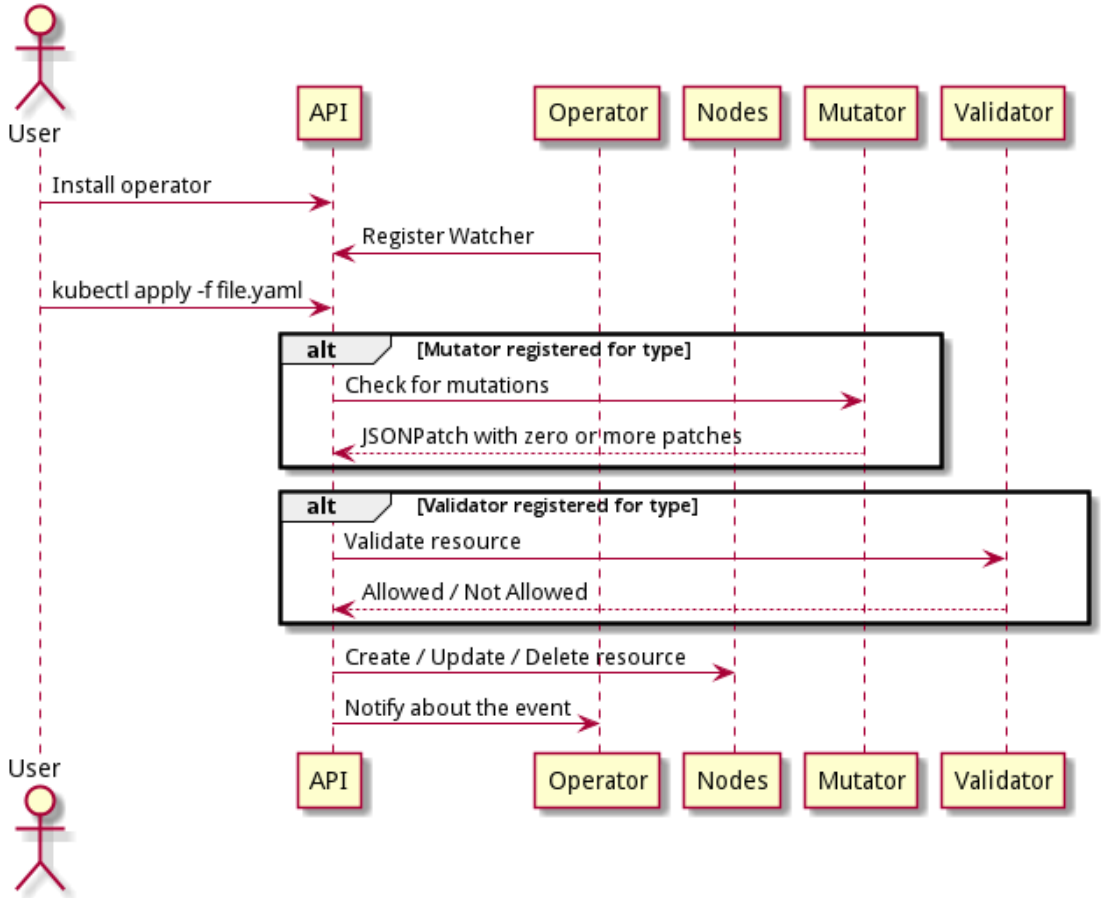

Figure 2.2: Kubernetes Operator Workflow

2.2.4 Sidecar

According to Burns and Oppenheimer (2016), the sidecar pattern is the most common pattern for multi-container deployments (Burns and Oppenheimer 2016, sec. 4.1). 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 collects log files written to the file system and forwards them towards some log processing software (Burns and Oppenheimer 2016, sec. 4.1). Another example is the Google CloudSQL Proxy[6], which provides access to a CloudSQL instance from a pod without routing the whole traffic through Kubernetes services.

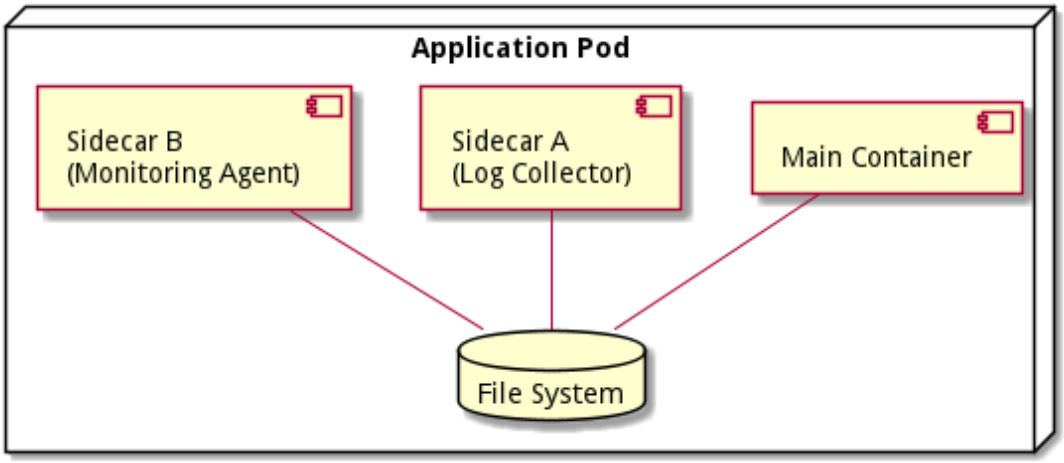

Figure 2.3: Sidecar container extending a main container in a pod. As example, this could be a log collector (Burns and Oppenheimer 2016, fig. 1).

The example shown in Figure 2.3 is extensible. Such sidecars may be injected by a mutator or an operator to extend functionality.

Common usecases for sidecars are controlling the data flow in a cluster in service mesh, providing access to secure locations 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 possible to scale a sidecar without scaling the Pod - and therefore the application - itself.

2.2.5 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 (Li et al. 2019, sec. 2). An example from the practice is "Istio"[7]. When using Istio, the

[6] https://github.com/GoogleCloudPlatform/cloudsql-proxy
[7] https://istio.io/

applications do not need to know if there is a service mesh installed or not. Istio will inject a sidecar (see Section 2.2.4) into pods and handle the communication with the injected services.

The service mesh provides a set of features (Li et al. 2019, sec. 2):

- **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 (Montesi and Weber 2016, sec. 2).
- **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.

2.3 Authentication and Authorization

2.3.1 Basic (RFC7617)

The `Basic` authentication scheme is a trivial authentication that accepts a username and a password encoded in Base64. To transmit the credentials, a construct with the schematics of `<username>:<password>` is created and inserted into the http request as the `Authorization` header with the prefix `Basic` (Reschke 2015, sec. 2). An example with the username `ChristophBuehler` and password `SuperSecure` would result in the following header:

```
Authorization: Basic Q2hyaXN0b3BoQnVlaGxlcjpTdXBlclNlY3VyZQ==
```

2.3.2 OpenID Connect (OIDC)

OpenID Connect is not defined in a RFC, the specification is provided by the OpenID Foundation (OIDF). OIDC however, builds on top of OAuth, which is defined by **RFC6749**.

OpenID Connect is an authenticating scheme, that builds upon the `OAuth 2.0` authorization protocol. OAuth itself is an authorization framework, that enables applications to gain access to a service (API or other) (Hardt and others 2012, abstract). OAuth 2.0 deals with authorization only and grants access to data and features on a specific application. OAuth by itself does not define *how* the credentials are transmitted and exchanged (Hardt and others 2012). OIDC adds a layer on top of OAuth 2.0 that defines *how* these credentials must be exchanged. This adds login and profile capabilities to any application that uses OIDC (Sakimura et al. 2014).

When a user wants to authenticate himself with OIDC, one of the possible "flows" is the "Authorization Code Flow" (Sakimura et al. 2014, sec. 3.1). Other possible flows are the "Implicit Flow" (Sakimura et al. 2014, sec. 3.2) and the "Hybrid Flow" (Sakimura et al. 2014, sec. 3.3). Figure 2.4 depicts the "Authorization Code Flow." A user that wants to access a certain resource 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 to 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) (Sakimura et al. 2014, sec. 2), the `access_token` can be in any format (Sakimura et al. 2014, sec. 3.3.3.8).

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

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.
SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

The stated JWT token contains:

```
{
  "alg": "HS256",
  "typ": "JWT"
}

{
```

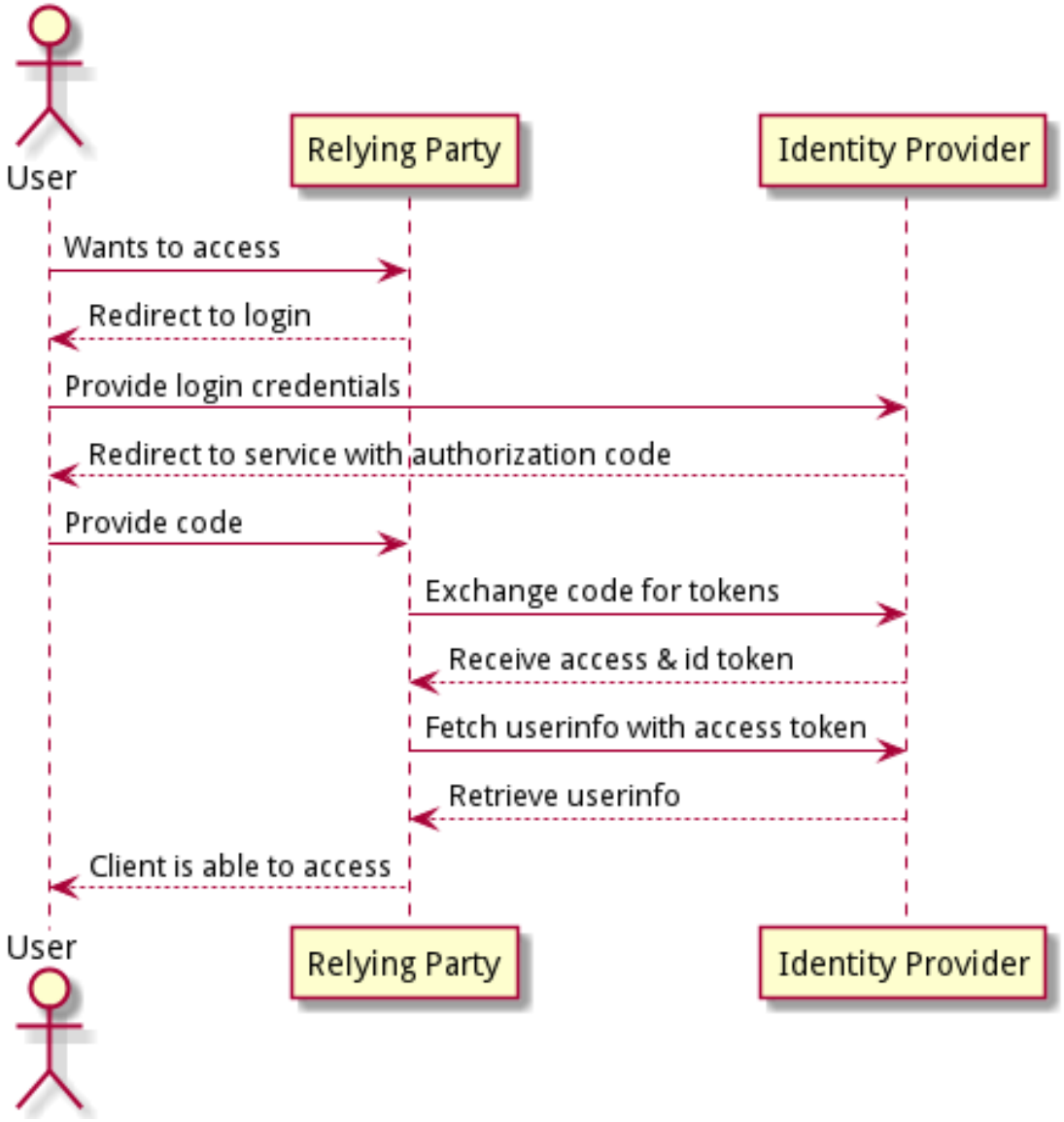

Figure 2.4: OIDC code authorization flow (Sakimura et al. 2014). Only contains the credential flow, without the explicit OAuth part. OAuth handles the authorization whereas OIDC handles the authentication.

```
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

3 State of the Art, the Practice and Deficiencies

This section gives an overview over the current state of the art, the practice, as well as the deficiencies to a desired situation.

3.1 State of the Art

In cloud environments, a problem that is solved is the transmission of data from one point to another. Kubernetes, for example, uses "Services" that provide a DNS name for a specified workload. For service meshes, additionally a sidecar is injected into the Pod that contains - in the case of "Istio" - an Envoy proxy to handle data transmission.

In terms of authentication and authorization, there exist a variety of schemes that enable an application to authenticate and authorize their users. OpenID Connect (OIDC) (see Section 2.3.2) is a modern authentication scheme, that builds upon OAuth 2.0, that in turn handles authorization (Sakimura et al. 2014).

Modern software architectures that are specifically designed for the cloud are called "Cloud Native Applications" (CNA). Kratzke and Peinl (2016) define a CNA as:

> "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." (Kratzke and Peinl 2016, sec. 3).

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 complexity, legacy applications and monoliths are not refactored or re-written before they are deployed into a cloud environment. If the legacy applications are mixed with modern systems, then the need of "translation" arises. Assuming, that the modern part is a secure application, that uses OIDC to authorize its users and the application needs to fetch data from the legacy system that does not understand OIDC, code changes must be made. Following the previous assumption, the code changes will likely be introduced into the modern application, since it is better maintainable and deployable than the

legacy monolith. Hence, the modern application receives changes that may introduce new bugs or security vulnerabilities.

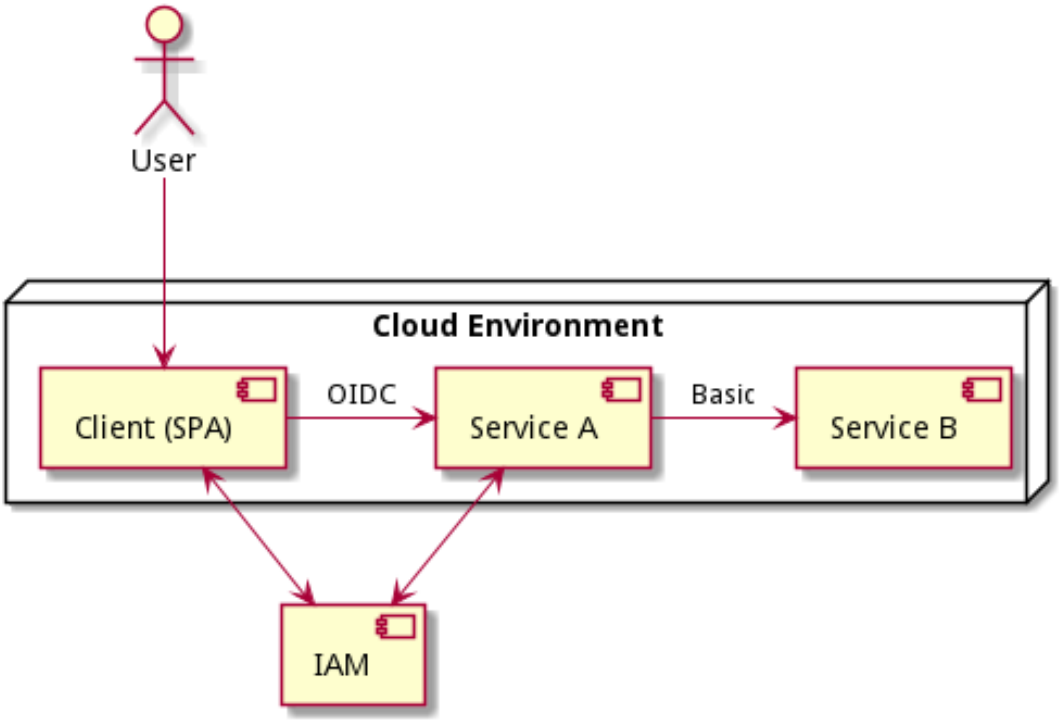

Figure 3.1: Microservice Architecture that contains modern applications as well as legacy services.

We consider the components in Figure 3.1:

- **User**: A person with access to the application
- **Client**: A modern single page application (SPA)
- **IAM**: Identity Provider for the solution (does not necessarily reside in the same cloud)
- **Service A**: A modern API application and primary access point for the client
- **Service B**: Legacy service that is called by service a to fetch some additional data

In the practice, we encountered the stated scenario at various points in time. Legacy services may not be the primary use-case, a nother case is the usage of third party applications which only support certain authentication mechanisms and the code is not accessable.

The process in Figure 3.2 shows the process of communication in such a described scenario. In Figure 3.2, the “Client” is the single page application (SPA), that authenticates against an arbitrary Identity and Access Management System (IAM). “Service A” is the modern backend that supports the client as backend API. Therefore, “Service A” provides

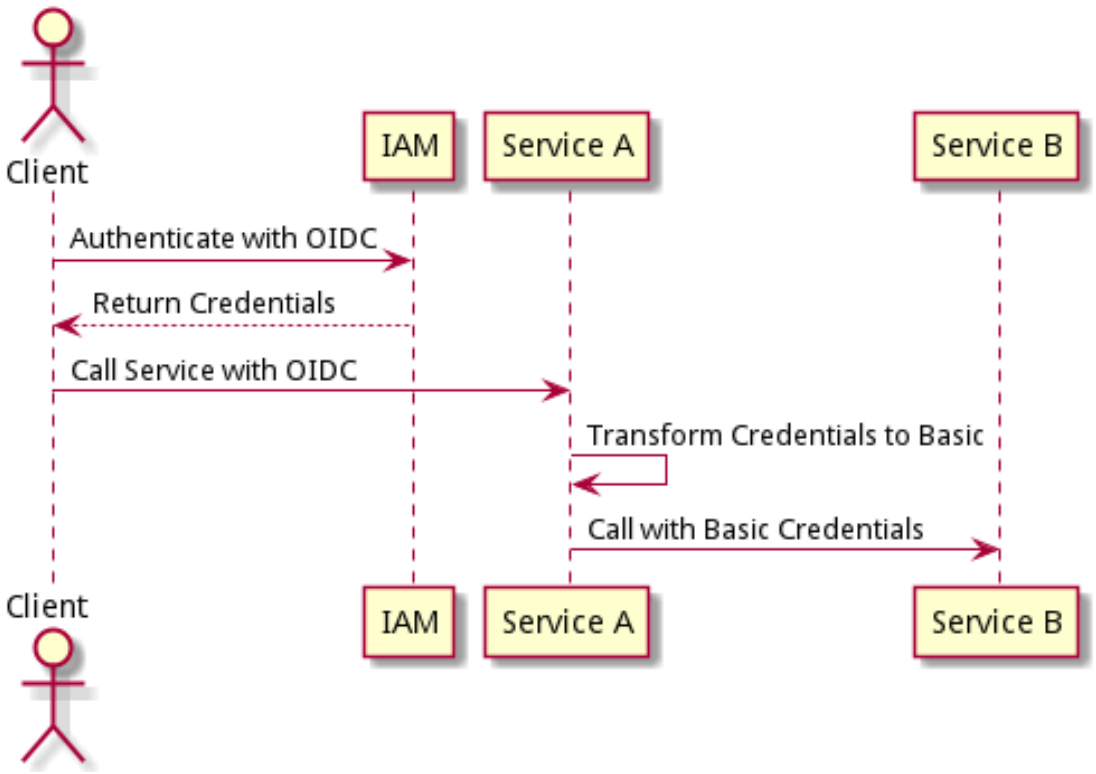

Figure 3.2: Current state of the art of accessing legacy systems from modern services with differing authentication schemes.

functionality for the client. “Service B” is a legacy application, for example an old ERP with order information, that was moved into the cloud, but is not refactored nor rewritten to communicate with modern authentication technologies.

In this scenario, the client calls some API on “Service A” that then will call “Service B” to get additional information to present to the user. Since the client and “Service A” communicate with the same authentication technology, the call is straight forward. The client authenticates himself and obtains an access token. When calling the service (“Service A”), the token is transmitted and the service can check with the IAM if the user is authorized to access the system. When “Service A” then calls “Service B” for additional information, it needs to translate the user provided credentials to a format that “Service B” understands. In the provided example, “Service B” is only able to handle Basic Authentication, as explained in Section 2.3.1. This means, if “Service A” wants to communicate with “Service B,” it must implement some translation logic to change the credentials to a format that B understands. This introduces code changes to “Service A,” since “Service B” is a legacy application that is not maintainable.

3.2 The Practice

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) (Istio Authors 2021b) as well as an external mechanism to provide custom authentication and authorization capabilities (Istio Authors 2021a). This 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.

3.3 Deficiencies

The situation described in the previous sections introduces several problems. It does not matter if “Service B” is a third party application to which no code changes can be applied to, or if it is a legacy application that cannot be updated for the time being. Most likely, the code change to provide the ability to communicate will be introduced into “Service A.” This adds the risk of errors since new code must be produced, which would not be necessary if the legacy service would be refactored. Also, changing “Service A” to communicate with B may be a feasible solution in a small setup. But as the landscape of the microservice architecture grows, this solution does not scale well. The matrix problem X services $* Y$ authentication methods describes this problematic. As the landscape and the different methods of authentication grows, it is not a feasible solution to implement each and every authentication scheme in all the services.

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 communication is between data centers, the communication and the credentials must be protected. It is not possible to create a zero trust environment with the need of knowledge about the targets authentication schemes.

The usage of a service mesh to mitigate the problem is not an option since the initial problem of transforming credential still persists. 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. Furthermore, service meshes introduce configurational complexity to the system which, in our opinion, is not needed without a clear usecase for a service mesh.

4 Distributed Authentication Mesh

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

The proposed architecture may be used as generic description for a solution to the described problem. For this project, the solution is implemented specifically to work within a Kubernetes cluster. The delivery of this project is a proof of concept to provide insights into the general topic of manipulating HTTP requests in-flight.

4.1 Definition

The solution to solve the stated problems in Section 3.3 must be able to transform arbitrary credentials into a format that the target service understands. For this purpose, the architecture contains a service which runs as a sidecar among the target service. This sidecar intercepts requests to the target and transforms the Authorization HTTP header. The sidecar is - like in a service mesh - 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 a well solved problem. In the given work, an Envoy proxy is used to deliver data between the services. Envoy allows the usage of an external service to modify requests in-flight.

4.2 Requirements

In Table 4.1, we present the list of requirements (REQ) for the proposed solution.

Table 4.1: 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 is injected as a sidecar into the solution. In Kubernetes this is done via an operator.

Name	Description
REQ 3	Beside the translator, an Envoy proxy is injected to the service inquestion to handle the data flow. This injection is also performed by the operator.
REQ 4	Translators do only modify HTTP headers, they do not interfere with the data that is transmitted. Any information that needs to be forwarded must be within the HTTP headers.
REQ 5	
REQ 6	

It is important to note that the implemented proof of concept (PoC) will not meet all requierements. Further work is needed to implement a solution according to the architecture that adheres the stated requirements.

4.3 Non-Functional Requirements

In Table 4.2, we show the non-functional requirements (NFR) for the proposed solution.

Table 4.2: 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 with security issues as stated in the OWASP Top Ten (https://owasp.org/www-project-top-ten) in mind.
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.
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 extend 100ms.
NFR 5	The solution is modular. It can be extended with additional "translators" which provide the means of transforming the given credentials to other target formats.

Name	Description
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 elements.
NFR 7	The architecture must be scaleable. The provided software must be able to scale according to the business needs of the overall system.
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

Like the requirements in Table 4.1, the PoC will not meet all NFRs that are stated in Table 4.2. Further work is needed to complete the PoC to a production ready software.

4.4 Contrast

To distinguish this solution from other software, this sections gives a contrast to two specific topics. The given topics stand for a general architectural idea and the contrast to the presented solution.

4.4.1 SAML

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 (Naik and Jenkins 2017). SAML is an XML framework for transmitting user data, such as authentication, entitlement and other attributes, between services and organizations (Naik and Jenkins 2017).

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 idea of SAML however, may be used as a baseline of security and the general idea of processing identities.

4.4.2 WS-*

The term "WS-*" contains a broad class of specifications within the WSDL/SOAP context. The specifications were created by the World Wide Web Consortium (W3C) but never finished and officially published.

The "Simple Object Access Protocol" (SOAP) is a protocol to exchange information between services in an XML encoded message (Curbera et al. 2002). It provides a way of communication between web services. A SOAP message consists of an "envelope" that contains a "body" and an optional "header" to transfer encoded objects (Curbera et al. 2002). An example SOAP message from Curbera et al. (2002) looks like this:

```
POST /travelservice
SOAPAction: "http://www.acme-travel.com/checkin"
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP:Envelope xmlns:SOAP="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP:Body>
    <et:eTicket xmlns:et="http://www.acme-travel.com/eticket/schema">
      <et:passengerName first="Joe" last="Smith"/>
      <et:flightInfo
        airlineName="AA"
        flightNumber="1111"
        departureDate="2002-01-01"
        departureTime="1905"/>
    </et:eTicket>
  </SOAP:Body>
</SOAP:Envelope>
```

The "Web Services Description Language" (WSDL), however, is an XML based description of a web service. The goal of WSDL is to provide a description of methods that may be called on a web service (Curbera et al. 2002). WSDL fills the needed endpoint description that SOAP is missing. While SOAP provides basic communication, WSDL defines the exact methods that can be called on an endpoint (Curbera et al. 2002).

The proposed solution differs from WS-* such that there is no exact specification needed for the target service. While the solution contains a common domain language - a SOAP like protocol to encode data - it does not specify the endpoints of a service. The solution merely interacts with the HTTP request that targets a specific service and transforms the credentials from the common format to the specific format. Of course, certain authentication schemes need specific information to generate their credentials out of the data.

4.5 Architecture

The following sections provide an architectural overview over the proposed solution. The solution is described in prosa text, as well as usual software engineering diagrams with explanations. First, a description of the solution gives an introduction about the idea, then the architecture shows the general overview of the solution followed by sequence and communication definitions.

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 a generalizable idea to implement such a solution, while the PoC proves the ability of modifying HTTP requests in-flight.

4.5.1 Brief Description

In general, when some service wants to communicate with another service and the user does not need to authenticate himself for every service, probably a federated identity is used. This means, that at some point, the user validates his own identity and is then authenticated in the whole zone of trust.

To achieve such a federated identity with diverging authentication schemes, the solution converts validated credentials to a common language format. 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 leak into the communication between services.

The basic idea of the solution is to remove any credentials from an outgoing HTTP request with the common format of the users identity and replace the common format in the ingoing HTTP request into the valid credentials of the given scheme.

4.5.2 Use Case

The usefulness of such a solution shows when “older” or monolythic software moves to the cloud or when third party software is used that provides no accessable source code.

Communicate with legacy software

Precondition: Cloud native application and legacy software are deployed with their respective manifests and the sidecars are running.

1. The user is authenticated against the CNA
2. The user tries to access a resource on the legacy software
3. The CNA creates a request and “forwards” the credentials of the user
4. The proxy intercepts the request and forwards the credentials to the transformer

5. The transformer verifies the credentials and transforms them into a common format
6. The proxy replaces the headers and forwards the request
7. The receiving proxy forwards the common format to the translator of the target
8. The translator casts the credentials into the specific authentication scheme credentials
9. The receiving proxy forwards the request with the updated HTTP headers

Postcondition: The communication has taken place and no credentials have left the source service (CNA). Furthermore, the legacy service does not know what credentials or what specific authentication scheme was used.

This use case can be changed such that the receiving service is not a legacy software but an old and non-maintained application that is deployed into a cloud environment without refactoring. Another possibility could be some third party application where the source code is not accessable.

4.5.3 Solution Architecture

In this section, we describe the system architecture of the proposed solution. The architecture is shown in a diagram and then broken down to the individual parts.

Figure 4.1 shows the general solution architecture. In the "support" package, general available elements are presented. The solution needs a public key infrastructure (PKI) to deliver key material for signing and validation purposes. Furthermore a configuration and secret storage must be provided.

Additionally, an optional automation component watches and manages applications. In case of cloud environments, this component is strongly suggested to automate deployment configuration. The automation does inject the proxies, translators and the specific needed configurations for the managed components.

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

For the further sections, the architecture shows elements of a Kubernetes cloud environment. The reason is to describe the specific architecture in a practical way. However, the general idea of the solution may be deployed in various environments and is not bound to a cloud infrastructure.

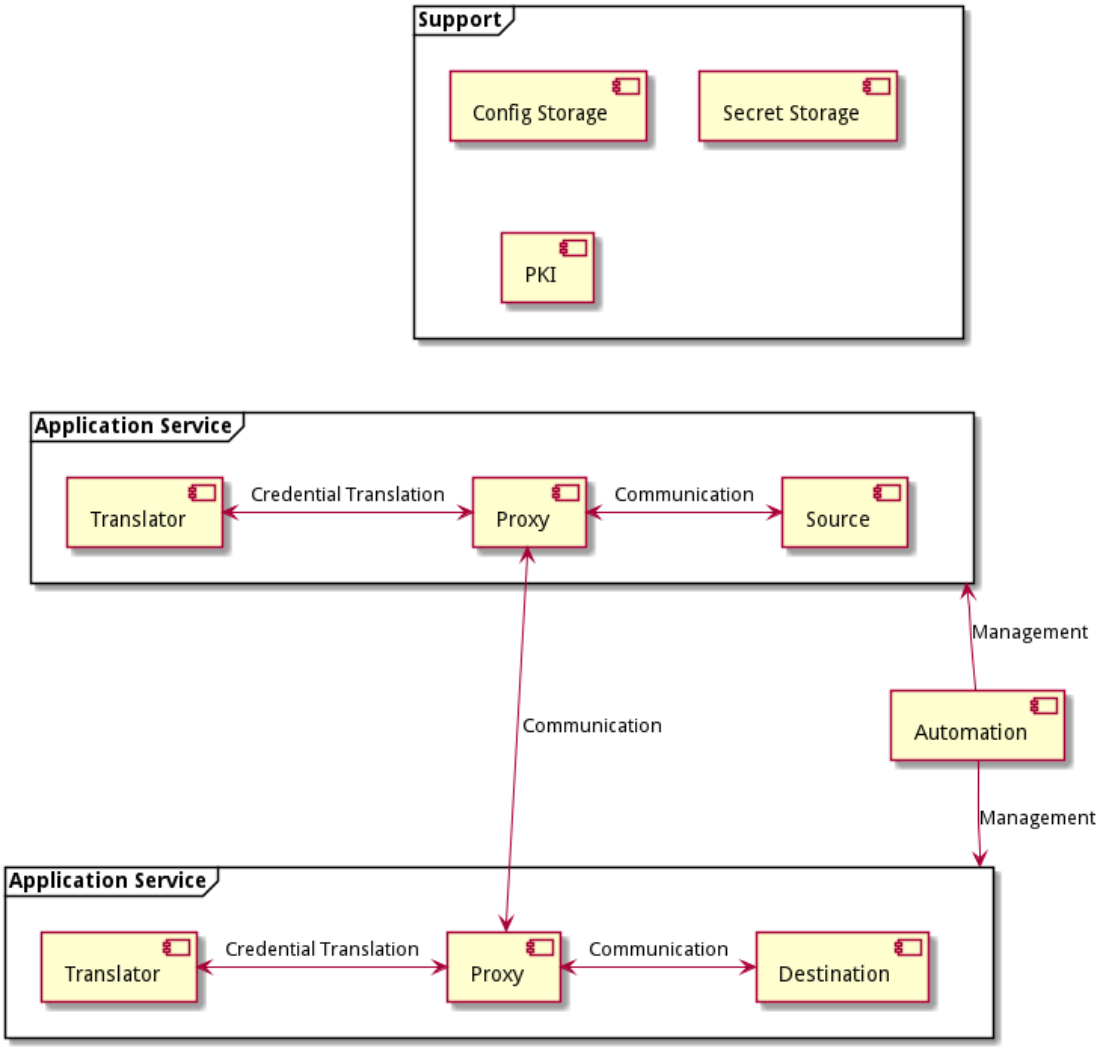

Figure 4.1: Solution Architecture

4.5.3.1 Automation

In case of a Kubernetes infrastructure, the automation part is done by an operator as explained in Section 2.2.3.

The operator in Figure 4.2 watches the Kubernetes API for changes. When deployments or services are created, the operator enhances the respective elements. “Enhancing” in this context means, that additional pods (see Table 2.1) are injected into a deployment as sidecars. The additional sidecars are the proxy and the translator. While the proxy manages incomming (“ingress”) and outgoing (“egress”) communication, the translator manages the transformation of credentials from and to a common format.

The process that enhances deployments is shown in Figure 4.3. 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 of the object in question “is relevant” by checking if it is 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, depending on the type, the operator injects sidecars into the deployment or reconfigures the service to use the proxy as targeting port for the service communication.

4.5.3.2 Public Key Infrastructure (PKI)

The role of the public key infrastructure in the solution is to build the trust anchor in the system.

Figure 4.4 depicts the relation of the translators and the PKI. When a translator starts, it aquires trusted key material from the PKI (for example with a certificate signing request). This key material is then used 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.

The sequence in Figure 4.5 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.

When communication happens, the proxy will forward the HTTP headers of the request to the translator which contains the transfered identity of the user in the DSL. In 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 may happen and the proxy forwards the communication to the destination.

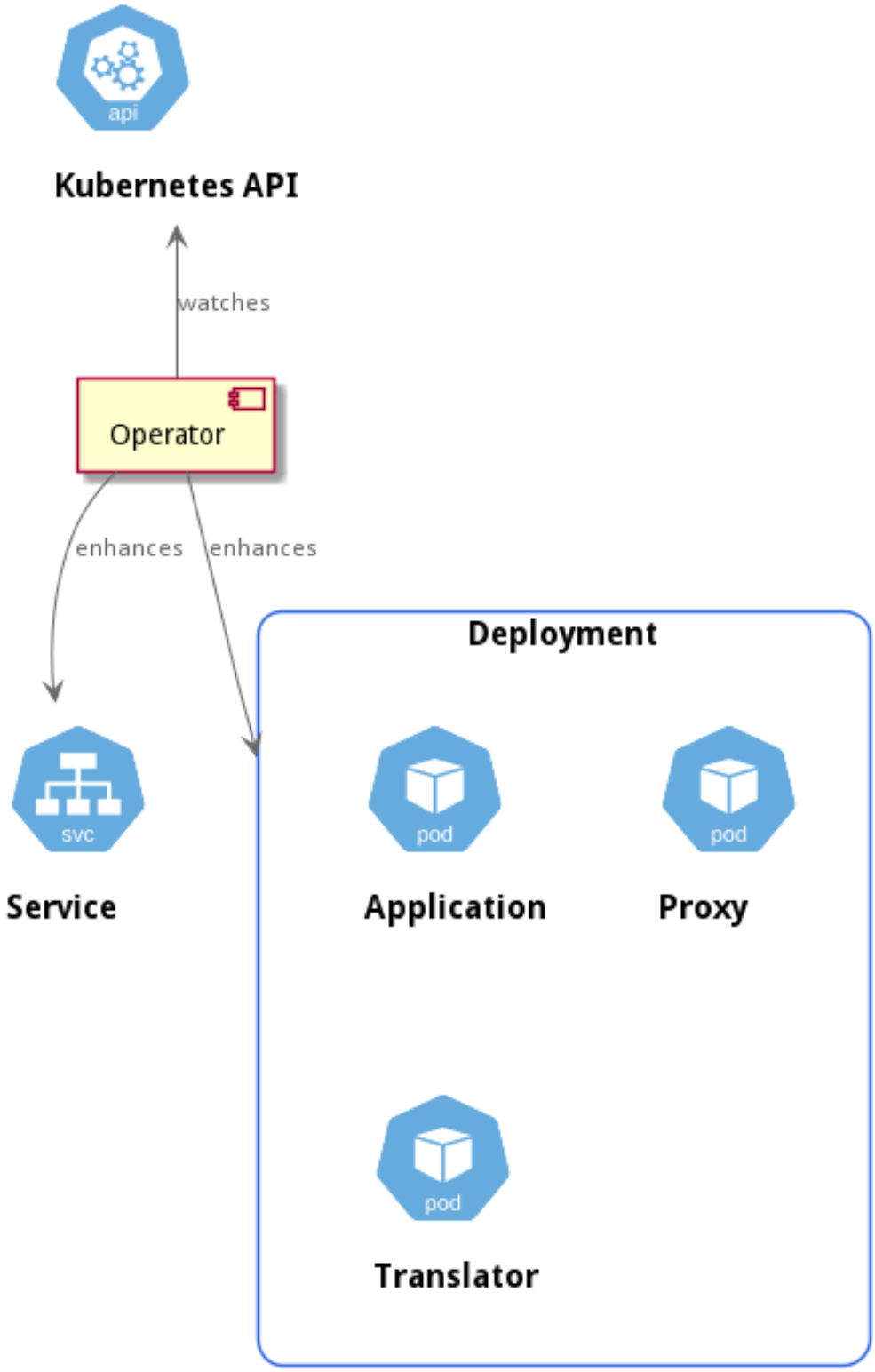

Figure 4.2: Automation Architecture

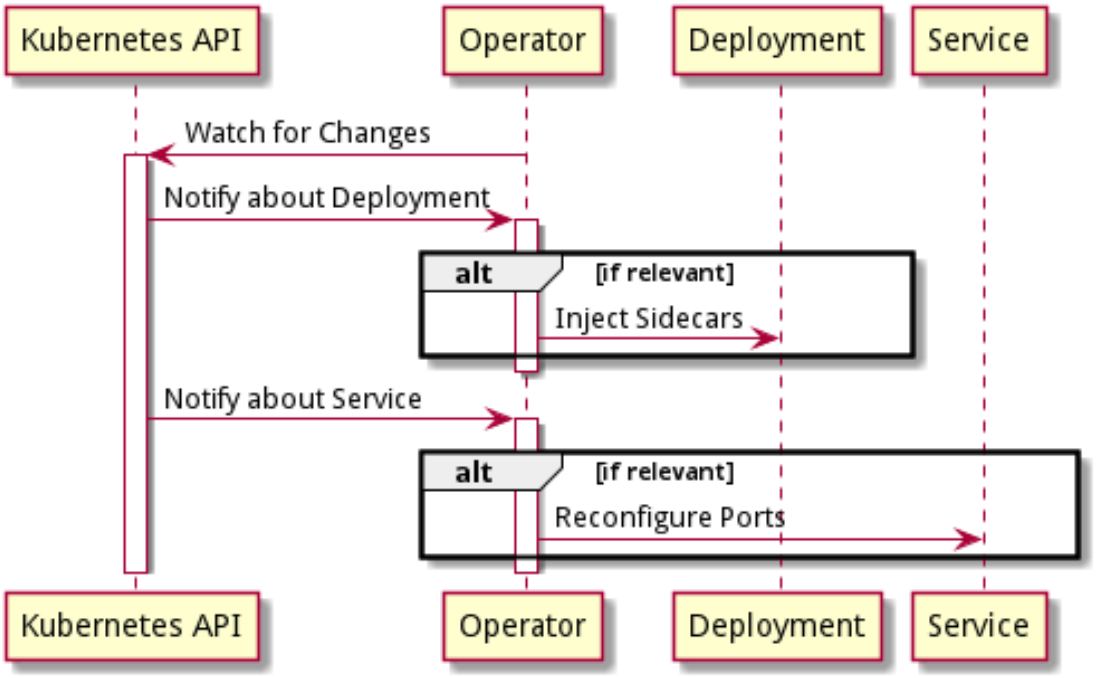

Figure 4.3: Automation Process

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

4.5.3.3 Networking

Networking in the proposed solution works with a combination of routing and communication proxing. 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.

As seen in Figure 4.6 the proxy is the mediator between source and destination of a communication. Furthermore, the proxy manages the translation by communicating with the translator to transform the identity of the authenticated user and transmit it to the destination where it gets transformed again. Additionally, with the aid of the PKI, the proxy can verify the identity of the sender via mTLS.

4.5.3.3.1 Ingress Figure 4.7 shows the general process during inbound request processing. When the proxy receives a request (in the given example by the configured Kubernetes service), it calls the translator with the HTTP request detail. The PoC

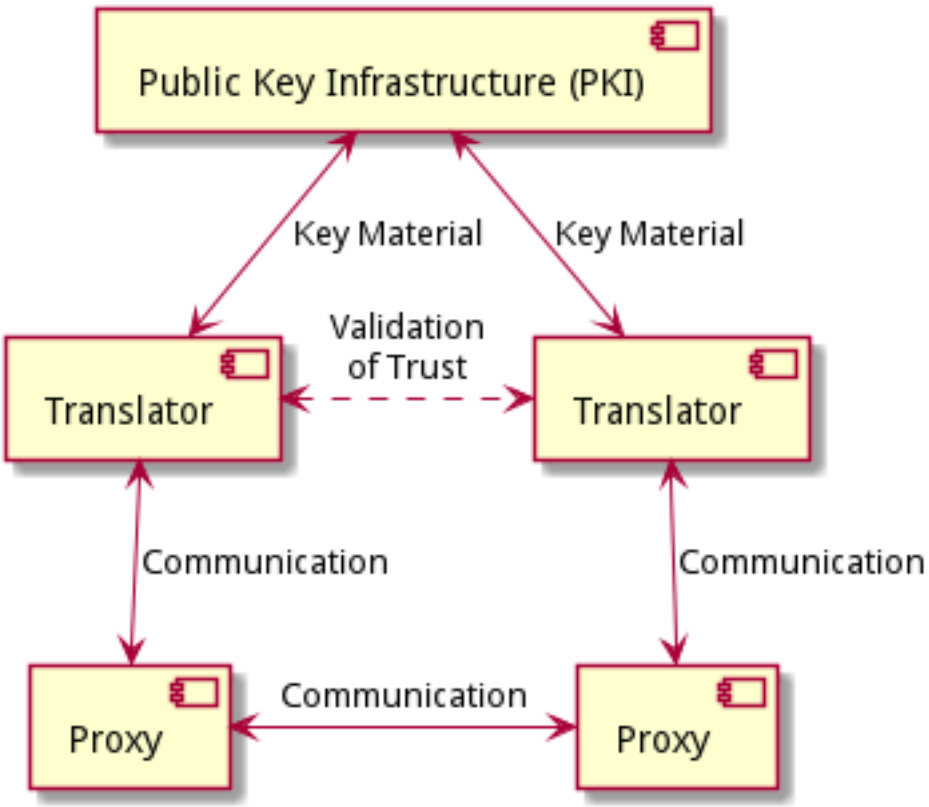

Figure 4.4: PKI Architecture

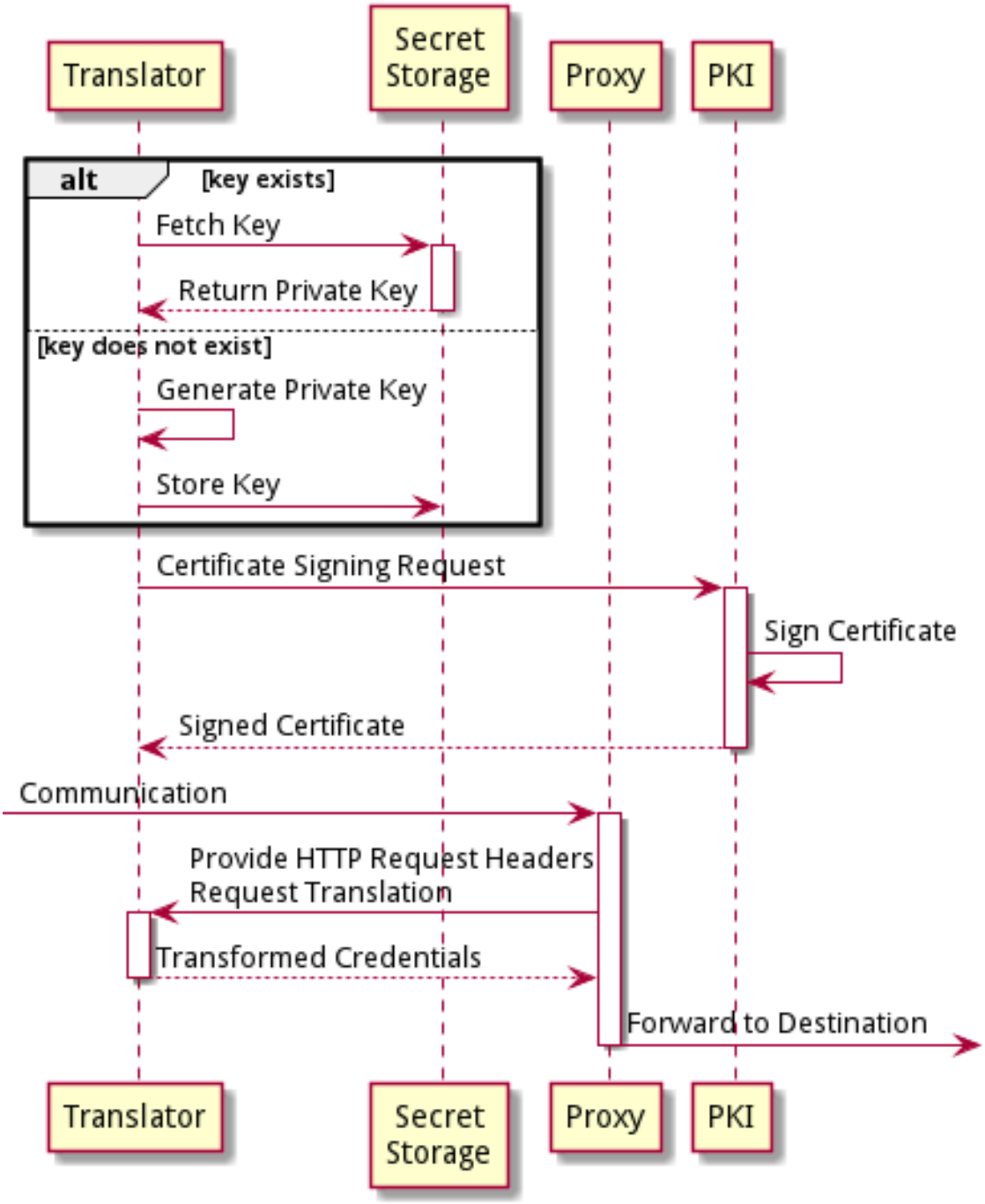

Figure 4.5: PKI Process

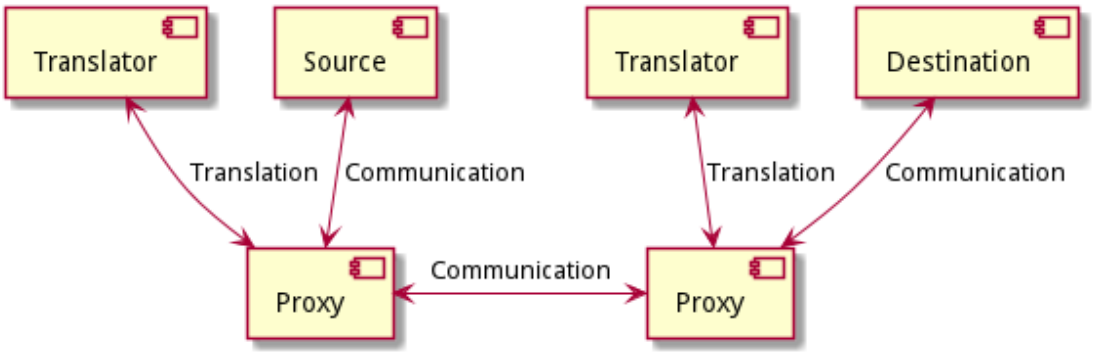

Figure 4.6: Networking Architecture

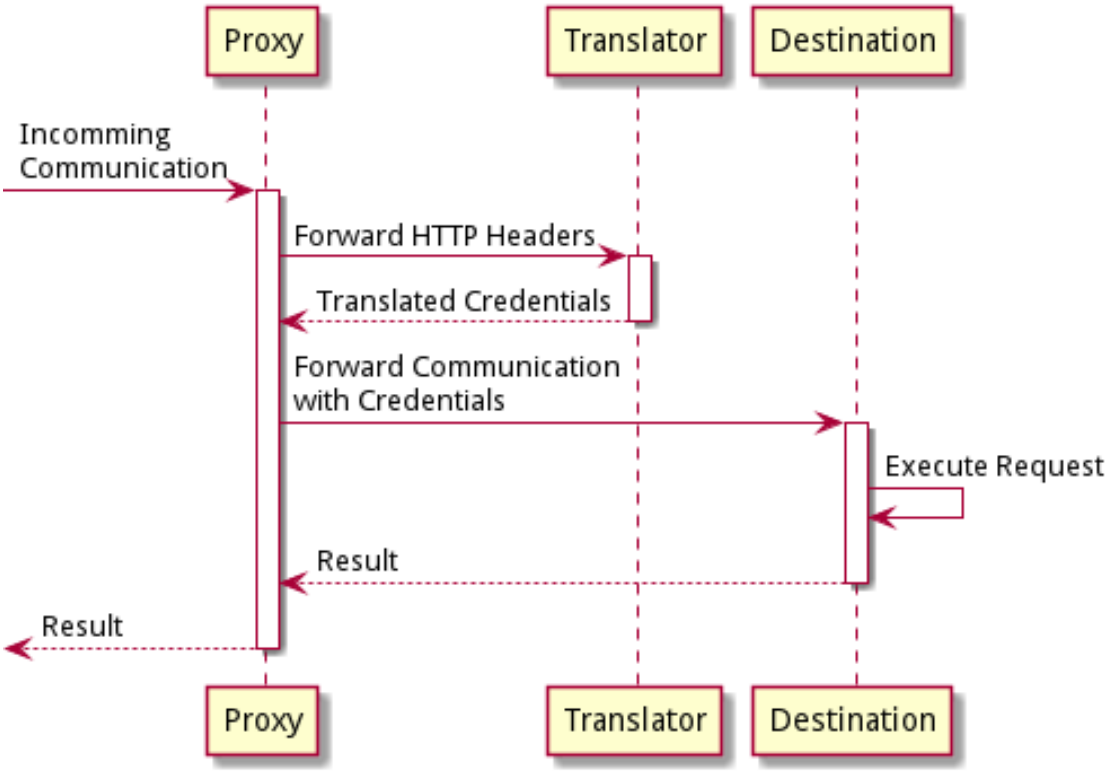

Figure 4.7: Inbound Networking Process

is implemented with the "Envoy" proxy. Envoy allows an external service to perform "external authorization"[1] 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 common format. 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 done by setting the `Authorization` header to static Basic Auth credentials.

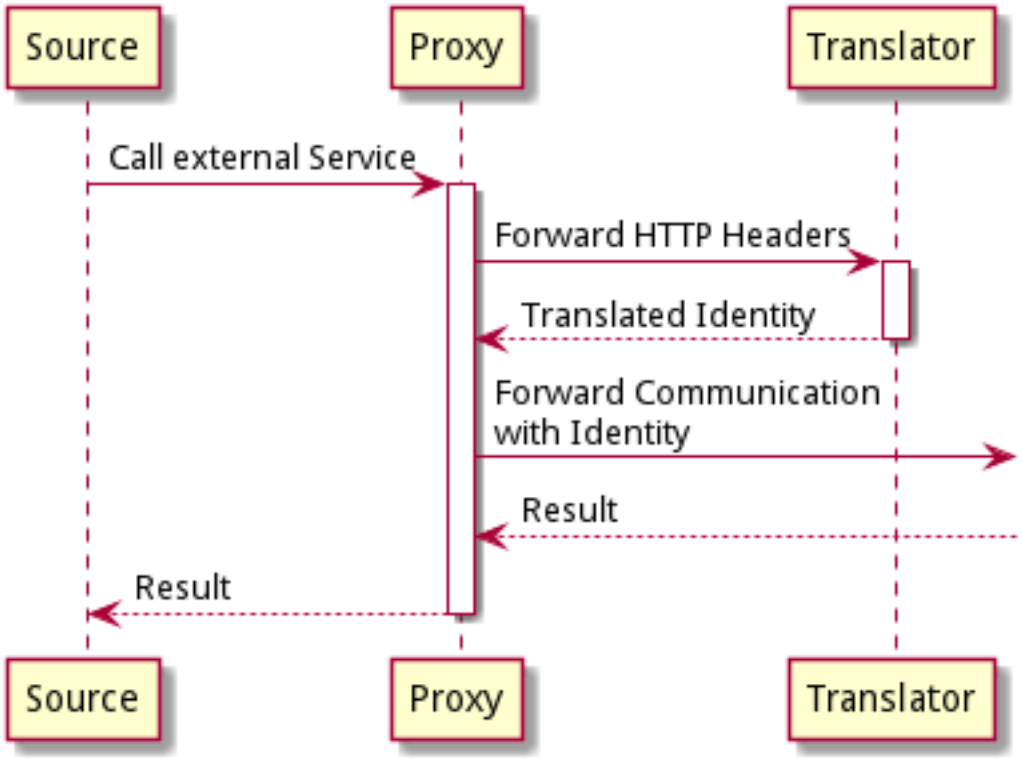

Figure 4.8: Outbound Networking Process

4.5.3.3.2 Egress In Figure 4.8 the outbound (egress) traffic is described. The proxy needs to catch all traffic from the source and performs the reversed process (of Figure 4.7)

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

by transforming the provided information from the source to generate the common format with the users identity. This identity is then inserted into the HTTP headers and sent to the destination. At the sink, the process of Figure 4.7 takes place - if the sink is part of the authentication mesh.

4.5.3.4 Translator

The translator is responsible for transforming the identity from and to the common domain specific language.

Figure 4.9: Translator Architecture

In conjunction with the PKI, the translator can verify the validity and integrity of the incomming identity.

When the translator receives a request to create the needed credentials, it performs the sequence of actions as stated in Figure 4.10. First, the proxy will forward the needed data to the translator. Afterwards, the translator will check 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.

In the PoC, the proof of integrity is not implemented, but the transformation takes place, where a "Bearer Token"[2] is used to check if the user may access and then replaces the token with static Basic Auth credentials.

4.5.4 Communication

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 contain the possibility to hash their contents and thus secure them against modification, JWT tokens are better designed for HTTP headers, since in current OIDC environments, JWT tokens are already used as access and/or identity tokens. They provide a secure environment with public and private claim names (Jones, Bradley, and Sakimura 2015, sec. 4.2, sec. 4.3).

[2]Opaque OIDC Token of an IDP.

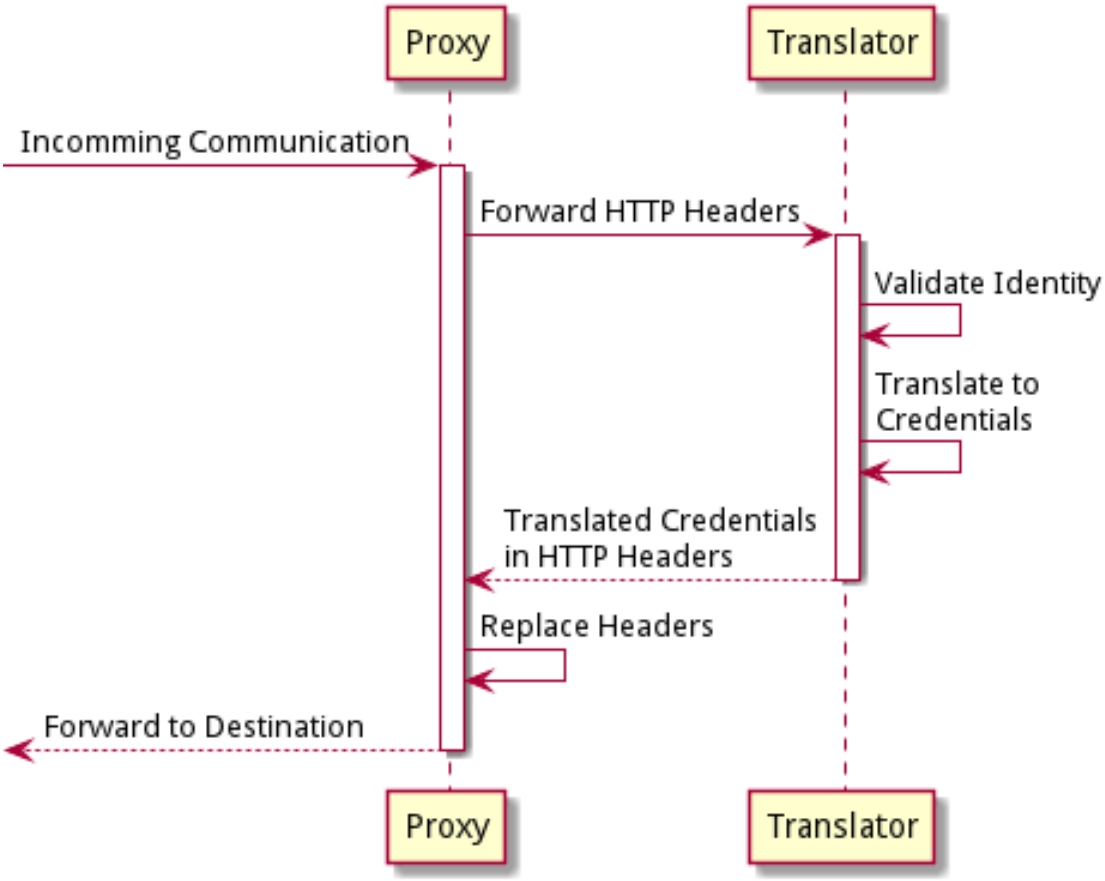

Figure 4.10: Translator Process

Other options could be:

- Simple JSON
- YAML
- XML
- X509 Certificates
- Any other structured format

The problem with other structured formats is that tamper protection and encoding must be done manually. JWT tokens provide a specified way of attaching a hashed version of the whole content (Jones, Bradley, and Sakimura 2015) and therefore provide a method of validating a JWT token if it is still valid and if the sender is trusted. If the receiving end has his 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** (Cooper et al. 2008) - introduce another valid way of transporting data and properties about something to another party. "Certificate Extensions" can be defined by "private communities" and are attached to the certificate itself (Cooper et al. 2008, sec. 4.2, sec. 4.2.2).

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 easily. Since this solution should be as easy to use as it can be, manipulating certificates in translators does not seem to be a feasible option. For the sake of simplicity and the well known usage, further work to this project will probably use JWT tokens to transmit the users identity.

4.6 Implementation Proof of Concept (PoC)

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

- 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"[3]
- "Proxy": Envoy proxy which gets the needed configuration injected as Kubernetes ConfigMap file

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

- "Translator": A .NET (F#) application that uses the Envoy gRPC defintions to react to Envoy's requests and poses as the external service for the external authorization
- "Showcase App": A solution of three applications that pose as demo case with:
  - "Frontend": An ASP.NET static site application that authenticates itself against "Zitadel"[4]
  - "Modern Service": A modern 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 2.3.1)

The PoC addresses the following questions:

- Is it possible 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 realized:

- Specify the concrete common domain language to transport identities
- Implement a secure way of transporting identities with validation of integrity
- Provide a production ready solution of some translators and the operator
- Integrate the solution with a service mesh
- Provide a production ready documentation of the solution
- Further investiage 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 common language format must be implemented. To be used in current cloud environments, an implementation in Kubernetes can provide insights on how to develop the solution for other orchestrators than Kubernetes.

4.6.1 Showcase Application

The showcase application is a demo to show the need and the particular usecase of the solution. The application resides in an open source repository under https://github.com /WirePact/poc-showcase-app.

[4] https://zitadel.ch

When installed in a Kubernetes cluster, the user can open (depending on the local configuration) the URL to the frontend application[5].

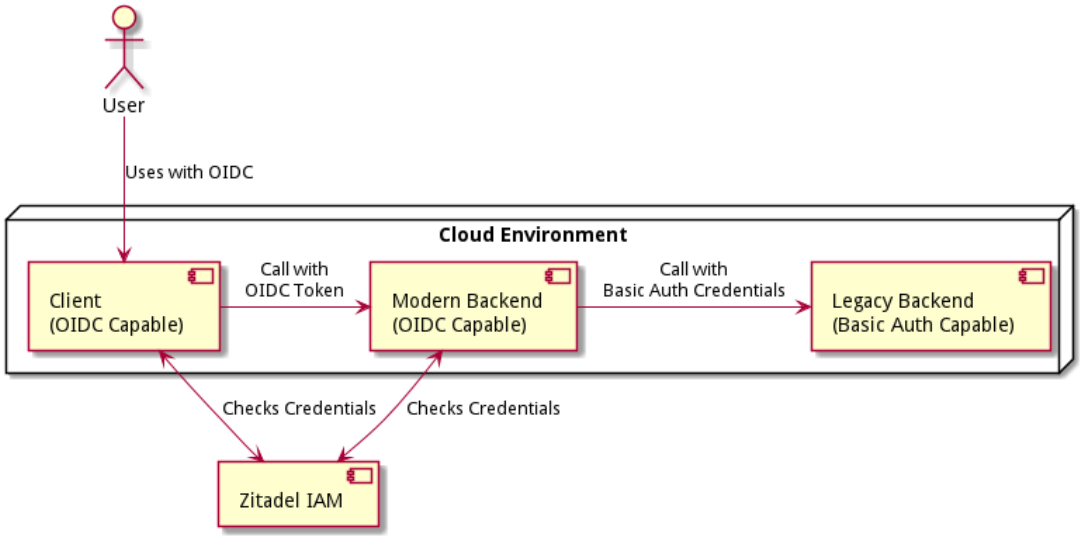

Figure 4.11: Component Diagram of the Showcase Application

Figure 4.11 gives an overview over the components in the showcase application. The system contains an ASP.NET Razor Page[6] 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 and the modern API is able to call an additional service on the legacy API.

In Figure 4.12, we show the process of a user call in the showcase 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 will call the modern backend API with the OIDC token and asks for customer and order data. The customer data is present on the modern API so it is directly returned. To query 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 one with either transformed basic auth credentials (when `USE_WIREPACT=false`) or with the presented OIDC token (otherwise). Either way, the legacy API receives basic auth credentials and returns the data which then in turn is returned and presented to the user.

[5]In the example it is https://kubernetes.docker.internal since this is the local configured URL for "Docker Desktop"

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

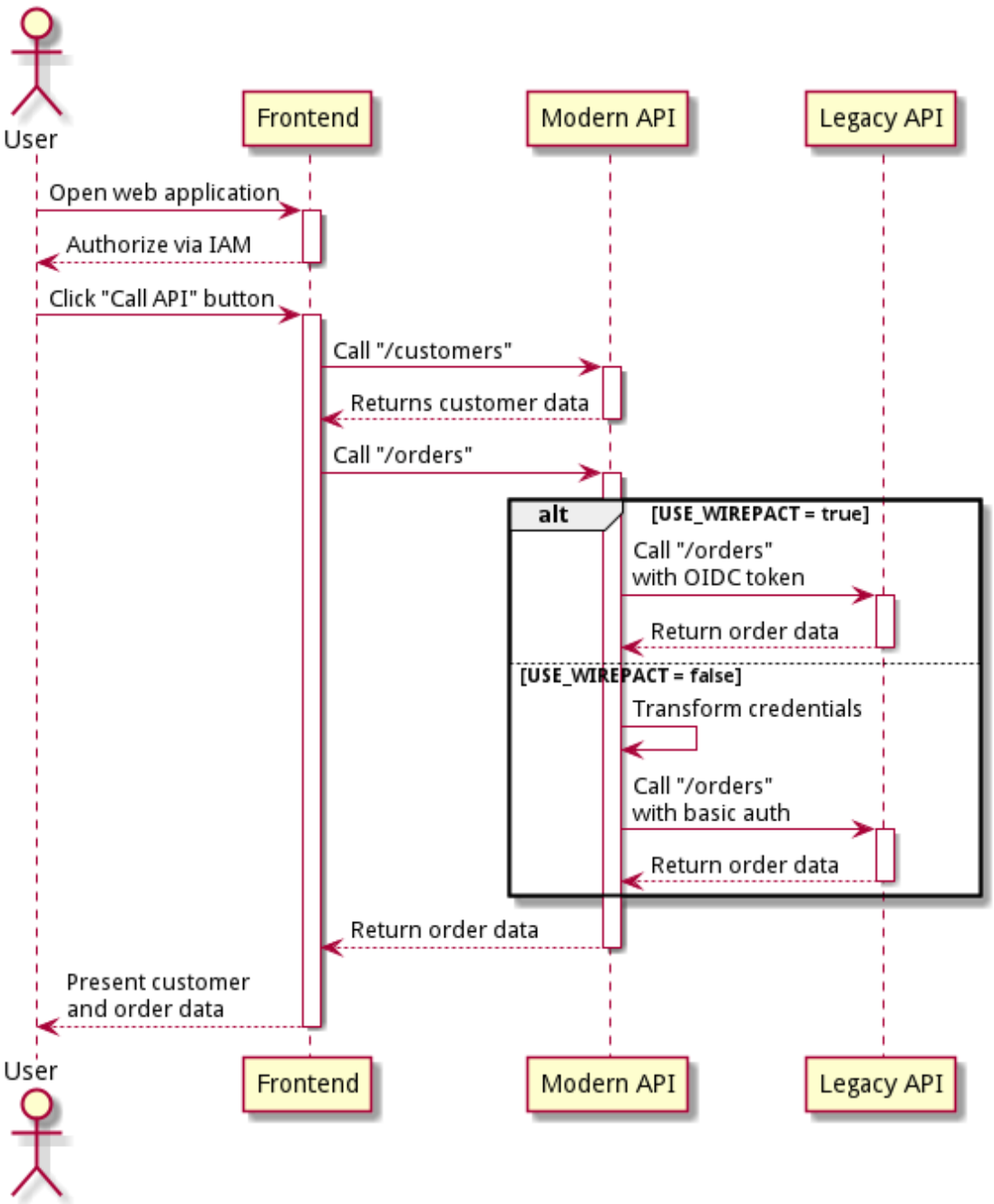

Figure 4.12: Sequence Diagram of the Showcase Call

To install and run the showcase application 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 solution, please refer to the installation guide in the Appendix.

4.6.2 Operator

TODO

4.6.3 Envoy Sidecar

In the PoC, the proxy sidecar is an Envoy proxy with an injected configuration. The operator injects the sidecar whenever a `Deployment` is created or updated via the Kubernetes API. The operator attaches the proxy and adds several annotations that are used for communication with a `Mutation Webhook`. Furthermore, a `ConfigMap` with the envoy configuration is created during the webhook.

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 an arbitrary request is allowed or not:

```yaml
# ... 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, the external authorization service must be added to the `clusters` list to be access via the configured name (`auth_translator`):

```yaml
# ... 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 gRPC is configured (`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

TODO

5 Evaluation

TODO: show in an evaluation, that the provided solution is working and improves the situation

6 Conclusion

TODO

6.1 Further Work

TODO

7 Bibliography

Bryan, Paul, and Mark Nottingham. 2013. “Javascript Object Notation (JSON) Patch.” RFC. Internet Engineering Task Force IETF. https://tools.ietf.org/html/rfc6902.

Burns, Brendan, and David Oppenheimer. 2016. “Design Patterns for Container-Based Distributed Systems.” In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. Denver, CO: USENIX Association. https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/burns.

CNCF. 2021. “Kubernetes Website.” *GitHub Repository*. https://github.com/kubernetes/website; GitHub.

Cooper, Dave, Sharon Boeyen, Stefan Santesson, Tim Polk, Russ Housley, and Stephen Farrell. 2008. “Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile.” 5280. Internet Engineering Task Force IETF. https://doi.org/10.17487/RFC5280.

Creative Commons. 2021. “Attribution 4.0 International (CC BY 4.0).” https://creativecommons.org/licenses/by/4.0/.

Curbera, F., M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana. 2002. “Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI.” *IEEE Internet Computing* 6 (2): 86–93. https://doi.org/10.1109/4236.991449.

Dobies, Jason, and Joshua Wood. 2020. *Kubernetes Operators: Automating the Container Orchestration Platform*. " O’Reilly Media, Inc.".

Hardt, Dick, and others. 2012. “The OAuth 2.0 Authorization Framework.” RFC. Internet Engineering Task Force IETF. https://tools.ietf.org/html/rfc6749.

Istio Authors. 2021a. “External Authorization.” *Istio*. https://istio.io/latest/docs/tasks/security/authorization/authz-custom/.

———. 2021b. “Mutual TLS Migration.” *Istio*. https://istio.io/latest/docs/tasks/security/authentication/mtls-migration/.

Jones, Michael B., Bradley John, and Nat Sakimura. 2015. “JSON Web Token (JWT).” RFC. Internet Engineering Task Force IETF. https://tools.ietf.org/html/rfc7519.

Kratzke, N., and R. Peinl. 2016. “ClouNS - a Cloud-Native Application Reference Model for Enterprise Architects.” In *2016 IEEE 20th International Enterprise Distributed*

Object Computing Workshop (EDOCW), 1–10. https://doi.org/10.1109/EDOCW.2016.7584353.

Li, W., Y. Lemieux, J. Gao, Z. Zhao, and Y. Han. 2019. "Service Mesh: Challenges, State of the Art, and Future Research Opportunities." In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, 122–25. https://doi.org/10.1109/SOSE.2019.00026.

Montesi, Fabrizio, and Janine Weber. 2016. "Circuit Breakers, Discovery, and API Gateways in Microservices." *CoRR* abs/1609.05830. http://arxiv.org/abs/1609.05830.

Naik, N., and P. Jenkins. 2017. "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)*, 163–74. https://doi.org/10.1109/RCIS.2017.7956534.

Reschke, Julian. 2015. "The 'Basic' HTTP Authentication Scheme." RFC. Internet Engineering Task Force IETF. https://tools.ietf.org/html/rfc7617.

Sakimura, Natsuhiko, John Bradley, Mike Jones, Breno De Medeiros, and Chuck Mortimore. 2014. "Openid Connect Core 1.0." Spec. The OpenID Foundation OIDF. https://openid.net/specs/openid-connect-core-1_0.html.

Appendix A: Installation Guide for PoC

TODO

Appendix B: Teaching Material for Kubernetes Operators

TODO